Preface

Lin Chao
Editor
Intel Technology Journal

Welcome to 1999, and the 1st Quarter issue of the Intel Technology Journal. The focus of this issue is Intel's current CAD tools, the tools used to design Intel's microprocessors. In the early 1960s, the process of designing the first integrated circuit was entirely manual. As one might guess, this was very expensive, laborious, and error-prone. However, in the 1970's, software CAD tools came to the rescue and became an integral part of the design of complex microprocessors produced by Intel.

These complex CAD tools are the focus of this issue. The first paper describes the architectural direction of Nike, Intel's next generation CAD tool suite, which supports code sharing to improve development efficiency, tool quality, and maintainability. As microprocessors approach deep sub-micron dimensions, the impact of physical design must be considered early in the circuit design phase to prevent costly re-designs. The second paper discusses Intel's FUB Circuit Design Environment combining circuit design with physical layout planning.

Datapath design (from RTL to layout) can take more than 60% of a project's human resources. In the third paper, a new design workflow is proposed along with a set of tools that will further automate the design process. The hope is that this proposed workflow will spur both academia and industry to tackle the problem of more automated datapath design tools. The fourth paper looks at the challenge Intel faces in making tools run on UNIX* and Windows NT* operating systems running on Intel and non-Intel architecture platforms. Tools and methods for formal verification at gate-level description are also discussed in the fifth paper. An example of formal functional verification of gate-level floating-point designs against IEEE-level specifications is presented. Structural and functional testing tradeoffs are described in the final paper. Again we feel that there are opportunities for the industry to provide more robust and scalable solutions for defect-based testing.

Designing for Success

By Greg Spirakis,
General Manager, Design Technology
Intel Corp.

At Intel, we deliver state-of-the-art microprocessors for every segment of the computer market. Our long history of innovation in Design Technology has enabled us to design, validate, and test each and every generation of these leading-edge microprocessors. This ability is among Intel's key competencies. It takes more than 100 software tools to design, validate, and test our microprocessors. These tools are either developed internally or procured from external vendors.

A major challenge ahead of us is the productivity gap identified by Sematech: the complexity of devices is increasing at more than double the rate of industry's ability to build them. This gap between process/manufacturing capabilities and design/test capabilities in the semiconductor industry continues to widen.

To address this gap, we challenged our Design Technology Division to address productivity as one of its major issues. Our target is to do twice as many products, in the same time-space, with the same size teams that we currently have. To this end, we developed the concept of providing a total solution to our engineering community that integrates internal and external tools, methods, and capabilities. For the next-generation products, this combination of technology, tools, and methodology is internally known as the Nike generation (preceeded by Athena, Zeus, and Cronus).

On the technical front, we are building an infrastructure that will keep our engineers productive while using multiple operating system environments such as Windows NT* and UNIX* on Intel 32-bit and future 64-bit hardware platforms. We also recognized the need to emphasize reuse in software engineering, architecture, and modeling, and therefore we developed a platform that allows developers across continents to share and integrate their solutions with as little overhead as possible.

For specific solutions such as reusing verification tools, designing datapaths, circuit design tools, and test strategy and tools, we defined productivity goals that will help us overcome the productivity gap mentioned earlier. Each tool has a specific productivity target that either reduces design time for the same problem or solves a more complex problem in the same amount of time.

In today's world of faster, smaller, and cheaper, our tools must meet the requirements of our next-generation microprocessors. In our Design Technology Division, as we move into the future, we will continue to develop the technologies that will provide Intel with a key competitive advantage over the next decade.

Nike's Software Architecture and Infrastructure: Enabling Integrated Solutions for Gigahertz Designs

V. Nagbhushan, Nike Development, DT, Intel Corp.
Yehuda Shiran, Nike Development, DT, Intel Corp.
Satish Venkatesan, Nike Development, DT, Intel Corp.
Tamar Yehoshua, Nike Development, DT, Intel Corp.

Index words: architecture, data model, software infrastructure

Abstract

This paper describes how Nike's innovative architecture addresses the expanding requirements of Intel's next-generation processor designs while enabling a design environment that is more productive than one built with the previous tool generation.

This paper shows how software architecture and data modeling techniques are used as core attributes of a CAD tool suite. We discuss the issues that have influenced Nike's architectural direction, such as technology trends, processor architecture trends, and computing platform trends. We identify some of the major drawbacks of existing tool suites and show how Nike architecture addresses these. Lastly, we describe how we developed a software infrastructure in order to support and facilitate the code sharing necessary to implement the designed architecture. The standard software development environment is described, including the tools and methodologies that are uniformly deployed to all Nike developers.

Introduction

Design and Test Technology (DT) is the supplier of CAD/CAE solutions for Intel's lead processor design projects. The Nike department within DT is chartered to provide the future generation CAD tool suite for use well into the next millennium. The first release of the Nike tool suite is scheduled for Q3, 1999.

Nike architecture was influenced by several external and internal vectors. The primary external vectors were industry technology trends, processor design trends, and Intel's design roadmap and computing platform trends. The primary internal vectors were the need to improve development efficiency, tool quality and maintainability, and to ensure adequate extendibility.

External Vectors

Industry technology trends predict a continuation of feature size reduction resulting in an exponential increase in chip transistor counts and a significant increase in frequency. Chips in the next decade could have upwards of 100 million transistors and run at frequencies well beyond 10GHz. As feature sizes decrease, aggressive circuit styles are also becoming the norm. This implies that second order effects, such as noise and inductance, become key factors in design decisions. In order to make these decisions effectively and efficiently, designers need increased visibility into data from multiple domains, such as circuit and layout.

Analysis of microprocessor architecture trends [1] and the Intel design roadmap show the emergence of highly integrated chips and ever decreasing time to market. This combination necessitates a design environment that will enable significant improvements in designer productivity.

A major change in the computing environment has been the emergence of Windows NT * on Intel® architecture (IA) as a sophisticated, inexpensive, and powerful platform. In comparison to the existing UNIX* environment commonly used for CAD, the Windows* environment provides a more consistent user interface (UI) as well as a rich set of office applications. The challenge for the developers of new CAD tool suites is to integrate the office environment with the CAD tools and use the Windows environment to provide a more productive system for the user.

* All other trademarks are the property of their respective owners.

Internal Vectors

As the coverage of the tool suite and the overall software size increases, improving software efficiency becomes very important. Software efficiency can be broadly categorized along two lines:

- *Extensibility and maintainability* of CAD tools. Even though individual tools might change, the core architecture of a tool suite persists for a very long time. For example, the previous CAD architecture at Intel spanned a decade. Given the dynamic nature of VLSI technology, it is impossible to predict accurately for the next decade. Hence, Nike architecture should be easily extendible and efficiently maintainable. For example, we need to build in enough headroom so that new manufacturing process features can be incorporated without massive system-wide code changes. Similarly, tools must be customizable to allow major changes in design methodology.
- *Tool quality and development efficiency.* In order to reduce the number of iterations in the design and implementation of complex software, there needs to be a well defined software development methodology where developers have detailed specifications and implementation plans upfront. In addition, quality needs to be built into the tools in order to avoid numerous cycles to fix defects.

In the next section we describe the Nike software architecture, goals, and principles. We then present a data-modeling methodology and architecture that are key enablers for achieving Nike architectural objectives. In the subsequent section, we describe a software infrastructure that supports and facilitates efficient development of Nike CAD tools.

Nike Software Architecture

This section describes a novel software architecture pioneered by Nike. We examine the drawbacks of existing CAD architectures. We then present LaMA, a layered modular architecture, followed by a set of architectural principles that drive the design and development of Nike.

Drawbacks of Existing Architectures

After studying existing CAD architectures, we found that they have several deficiencies. The following is a brief overview of the salient root causes of these deficiencies and the symptoms that they exhibit:

- *Non-modular*. Software was not written in a modular fashion. This often led to local implementations of similar or even identical functionality, resulting in inconsistent behavior. Users had to reconcile inconsistencies such as different timing or RC modules in various tools. It also made reuse difficult and severely impacted development efficiency. One of the causes of non-modularity was the existence of several data models; for example, multiple data representations for layout.
- *Difficult data exchange across domains*. Multiple data models in various domains were inconsistent in terminology, interfaces, and implementation. This made it difficult to provide a unified mechanism to map entities across the domains. Such mapping, when necessary, was done in an ad-hoc fashion that often resulted in loss of data, and consequently, productivity.
- *Ad hoc persistence mechanisms*. Typically, ASCII files were used to store data persistently. However, there was inadequate use of industry standard formats, often leading to a proliferation of files, multiple readers/writers, and semantic mismatches between formats representing the same data. ASCII files are also not performance oriented, have fundamental capacity limitations, are intolerant of new software releases, and make it very difficult to implement incremental input/output.
- *Inconsistent look and feel*. Users need to familiarize themselves with different interfaces to perform similar actions. This results in a high learning curve as well as loss of productivity.

Layered Modular Architecture (LaMA)

Nike is pioneering Layered Modular Architecture (LaMA), a hierarchical decomposition of a complex tool suite consisting of over a hundred tools with a total code size exceeding a million lines of code. Figure 1 shows the basic LaMA pyramid.

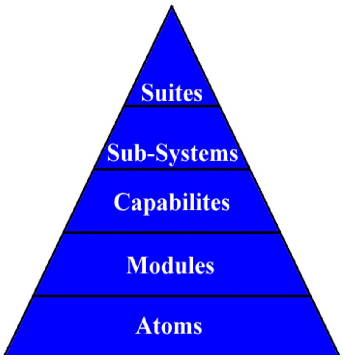

Figure 1: Layered modular architecture

In this model, the overall tool suite consists of several sub-systems; each sub-system is targeted at a particular user flow. A sub-system is comprised of sev-

eral capabilities each of which represents a user-visible functionality. A capability is implemented by one or more software modules, which are in turn made up of several atoms. The modules and atoms are designed with well-defined interfaces to facilitate reuse by several capabilities.

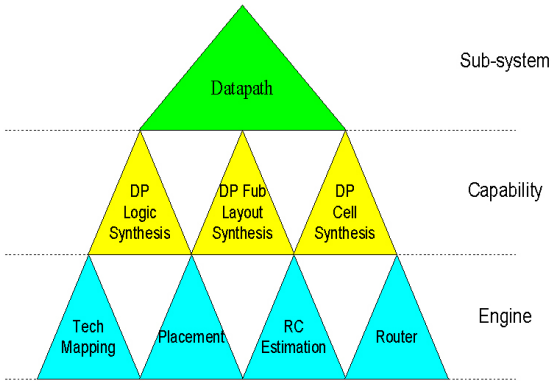

Figure 2: Example of LaMA sub-system

Figure 2 shows an example sub-system and its decomposition. Users may interact with these sub-systems through a visual cockpit, which enables simultaneous interaction with multiple domains such as circuit and layout. This reduces the number of design iterations caused by downstream surprises. The modular architecture enables a software developer to reuse the modules and atoms, and the cockpit integrator to reuse the capabilities to the maximum extent.

Software Architecture Principles

The ultimate objective of Nike software architecture is to enable efficient development of the CAD software that can meet or beat the high-level specifications set forth for the system. In order to drive this objective, we formulated a set of architectural principles. They were kept general enough so as to apply to all areas of the Nike tool suite; functional and area-dependent factors (e.g., principles that may apply only to timing tools) are not covered here. These main architectural principles are briefly explained below:

- Provide *integrated tool suites (or cockpits)* that enable users to execute a flow or perform similar operations. The primary goal is to improve user productivity.
- Software components should have a *modular design* and should be implemented *using common, unified services*. Each component should have a well documented interface, and similar functional components across the entire Nike tool suite should be implemented as common services. For example, there should only be one RC estimator for a given input abstraction, accuracy, and runtime. This can significantly improve reuse, quality, development efficiency and end-user consistency.
- The Nike system should present a *common look and feel*. The look (visual appearance) of UI objects that perform similar operations should be similar. The feel (behavior) of similar operations and data objects should be similar. The goal is to improve the predictability of the system, and consequently, the productivity of the user.
- Tools should support *incremental processing*, that is, they should be able to handle small delta changes to the input by incremental processing, and produce results that exhibit small delta changes.
- The Nike system should be *extensible.* Both, individual capabilities, as well as the entire tool suite should allow a user to easily extend its functionality.
- The Nike system should support *plug and play* with external tools.

A subsequent section describes a data modeling methodology and architecture that embodies these principles and enables us to achieve the architectural objectives. While a detailed description of the entire Nike architecture is beyond the scope of this paper, the following section describes the Nike layout architecture and how it meets the goals stated above.

Nike Layout Software Architecture

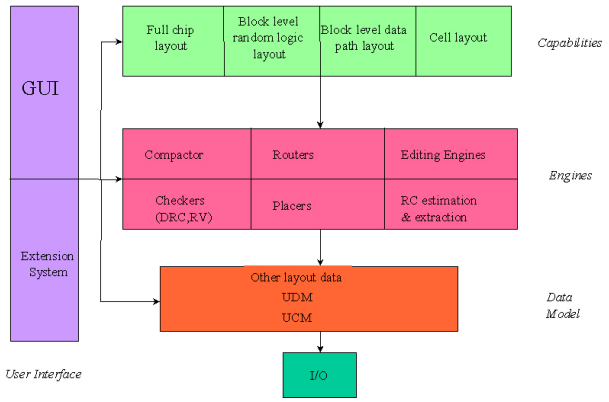

Figure 3: Nike layout architecture

Figure 3 shows a layered view of the Nike layout software architecture where most of the software modules have been partitioned into four layers: data model, engines, capabilities, and UI. A software module can, in general, access any module in a layer below it, either directly or indirectly (e.g., the full chip module

can make calls to any of the engines or data modules). Each module has a well defined procedural interface (API), which can be accessed by other modules.

The data model (DM) layer is the foundation of the architecture and consists of the software that models CAD data in memory and its interfaces. All the modules access CAD data by making calls to the DM APIs. The engines layer comprises mostly algorithmic modules (such as design rule checkers) that perform analysis and synthesis operations on the data. Modules in the capabilities layer address requirements of a subset of the domain (e.g., full-chip module). The user interface (UI) layer, drawn vertically, represents the user interface presented by all the other modules. The I/O module below the DM layer is a specialized engine to provide file input/output and persistence services. It accesses a slightly lower-level DM interface (compared to other engines) to enable fast I/O. The following paragraphs provide details about each layer:

- The *data model* layer serves as the in-memory repository for all the primary, non-derivable data in the system. In the case of layout, it is called the unified data model (UDM) and contains data (such as cells, wires, nets, transistors, etc.) and functions to access and modify the data. The data is modeled as a hierarchical class system, which is described in the next section. The API to this layer guarantees consistency of the data structures and semantics.
- The *engines* layer is comprised of software modules with well defined functionality. Algorithmic engines (such as RC estimation and extraction, placement, global and detailed routing (GR, DR), compaction, netlisters) and core editing engines (wire editing, move, etc.) fall into this layer. All the engines work off data from the DM layer, but may create temporary, derived data for efficiency. For example, the DRC engine works off scan-line data that is derived from the DM data. The derived application data may be saved along with the primary UDM data by the persistence mechanism to enable incremental processing.
- The capabilities layer is mainly comprised of environments that address the traditional sub-domains such as full-chip layout, block layout (for random logic and datapath) and leaf cell. Each of these provides functionality and customization appropriate to that capability. Each capability can either directly expose an engine functionality to the user (through the UI layer) or hide or modify it.
- The user interface (UI) layer provides user visibility into functionality and data. It is comprised mainly of a graphical user interface (GUI) and extension system. The capabilities and some engines instantiate GUI items at run-time to interact with the user. Engines such as DRC provide their own GUI to customize rules and display and scan errors. This enables them to be completely self-contained and reusable. The extension system enables customization by allowing easy access to data and functionality. The data model, engine and capability modules, provide access through a Tcl interpreter and through Windows* automation interfaces. GUI customization is enabled by VisualBasic* for Applications (VBA) on Windows NT*.

All the capabilities shown in Figure 3 are provided to the user in an integrated framework called the Nike Integrated Layout Environment (NILE).

The layered architecture ensures that there are no loops in the module dependency graph. Each module has a well defined API to maximize reusability; for example, the engines are reused by several capabilities. This re-usability has enabled us to develop new capabilities very quickly.

We have invested significant effort to ensure that each function is implemented by only one software module. For example, there is only one RC estimation module. If implementation of multiple modules began simultaneously, they were merged after the requirements matured.

The common GUI layer guaranteed a common look (e.g., native NT-looking widgets, docking windows, common list, and tree viewers, etc.). The I/O module is the single entry and exit channel for all design data. This has enabled us to minimize semantic mismatches.

Data Modeling

From a data modeling perspective, the processor design cycle can be divided into three phases: logic, circuit, and layout. Each phase successively refines the level of abstraction by adding details.

The design cycle involves many iterations, both within a phase as well as between phases. At each phase, a new representation is created, analyzed, and iteratively improved to meet system requirements. One objective of CAD tools is to minimize the time for each iteration and to minimize the total number of iterations to reduce time-to-market.

Figure 4 illustrates classic and modern design paradigms. In the classic paradigm, design phases are sequential in nature with the underlying assumption that optimal implementation of each phase results in an optimal realization of the next. In the modern design paradigm to be supported by Nike, all design phases proceed in parallel. Within each design phase, infor-

mation is required from the other two. This necessitates ease of interaction between design tools at different design phases.

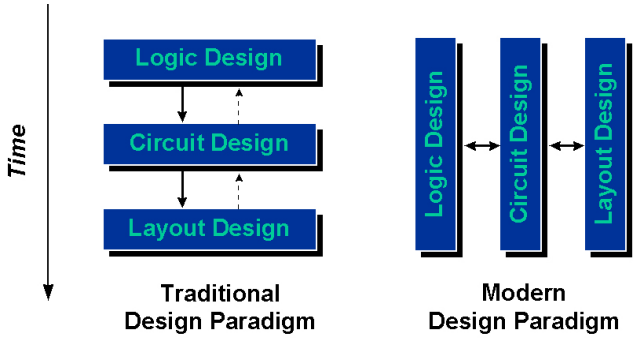

Figure 4: Design paradigms

Motivation

In terms of direct end-user benefit, data models are primarily driven by the productivity vector. This implies that data models should facilitate smoother automation/integration between different design tools with simpler work flows for certain tasks, and they should provide users with better capabilities to make tradeoffs/optimizations across design domains as well as between tools within the same domain.

These high-level requirements can be mapped into the following Nike system architecture goals:

- Enable a flexible configuration that facilitates a plug-and-play system architecture; different combinations of software components should be usable cooperatively. This is significant because system requirements will continue to evolve over the next several years.
- Achieve semantic consistency in data representation across layout, circuit, and logic domains. This will facilitates correct data transformations and exchange between tools in different domains.
- Promote reuse of common software components.
- Insulate individual CAD capabilities from persistent storage issues. This enables ease in changes to a storage mechanism as well as allowing multiple forms to exist transparently.

There is a trend in the EDA industry towards delivering a suite of inter-operable components rather than point tools. The next section presents a data model architecture that is helping us cope with this trend.

Data Model Architecture

As described above, the entire processor design cycle may be viewed as transformations between representations. At each representation, multiple CAD tools act as producers and consumers. A layout editor is an example of a design producer; simulators and design verification tools are examples of consumers. Data flow between producers and consumers typically transcends domain boundaries. For example, a layout routing tool may require information from a circuit timing engine; the timing engine in turn requires information from a layout parasitics calculator. Data modeling for the CAD domain is complicated by the diversity of design producers, consumers, and their singularities. This diversity also implies that different design tools need to view different aspects of a design. These considerations necessitate a modeling framework that can accommodate an assortment of data types and also be extensible to facilitate inclusion of new types. Our solution is to provide a framework that allows multiple levels of data models, related by specialization/generalization relationships, each tailored to the specific needs of applications. This facilitates interoperability and reuse of existing models. Semantic mappings between models promote ease of information
exchange. Figure 5 illustrates this modeling framework.

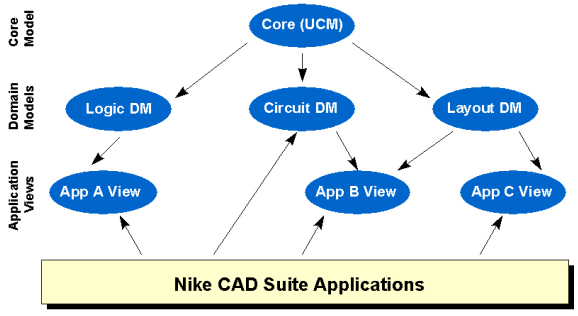

Figure 5: Data model framework

At the root of the framework is a unified core data model (UCM) that serves as the common vector between logic, circuit, and layout domains. It defines a

set of design entities that have consistent semantics across domains. These entities are primarily concerned with representing the hierarchy and connectivity[1] of a design, along with an interface to manipulate it. Domain models extend UCM with domain specific information. For example, the layout data model would add geometrical information. CAD applications can operate directly on UCM, a domain model, or an application view. An application view is either an extension of a data model with application-specific entities, or a suitably adapted perspective of the information in a data model. In all of the domain models and application views, UCM continues to be the common baseline. Having unified domain models allows the reuse of engines and minimizes I/O overhead. The reduction of the number of data models also facilitates the development of centralized cross-domain mapping services.

Figure 6 illustrates a usage scenario where UCM serves as a basis for enabling data-driven interactions between capabilities in the circuit and layout domains. Semantic mappings between data in the layout planner and circuit design environment are accomplished using a Mapper module. The Mapper has the ability to map from a design object in one domain to a corresponding design object in another domain; the mapping is performed at the granularity of UCM entities.

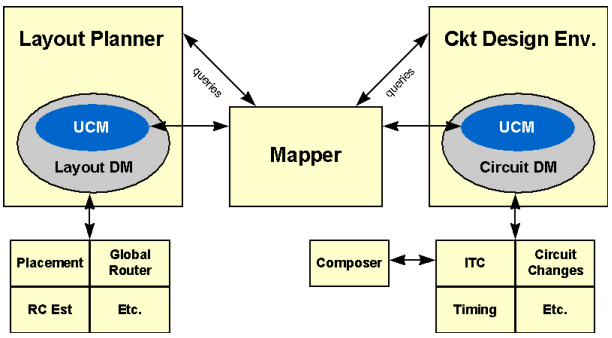

Figure 6: Sample usage scenario

Since the amount of time spent by each CAD tool on its task can be significant, it is also essential for interactions between tools to be incremental in nature. A designer must be able to operate on data in one domain and interactively see the effect of the change in another domain. As a concrete example, a designer can modify block placement in the layout planner and get interactive feedback on the effect of the new route on circuit timing. Although, data models alone cannot provide incremental interactions, they facilitate the development of modules and methodologies that can.

In summary, two key architectural goals have been addressed:

1. Facilitate interactive interactions across CAD tool boundaries.
2. Enable incremental iterations through the design cycle.

In the next section, we describe the software building blocks necessary to realize our vision of a modular architecture. We itemize the software tools and methodologies we implemented in Nike.

Software Infrastructure

Nike's architecture requires a strong focus on software quality. A common data model and a high level of module reuse introduce dependencies between projects that can magnify any software defects. Development at three sites adds additional complexity. To support high-quality development, Nike has defined and implemented a standard software development environment, including tools and methodologies, that are uniformly deployed to all Nike developers.

Standard Development Environment

The initial step was to itemize the software development tools being used across the department and set the Software Development Environment (SDE) Plan of Record (POR). The SDE POR lists all of the development tools and their versions that should be installed for each developer. This step is critical to ensure that all code libraries will be compatible and that common methodologies can be implemented.

Configuration Management

Our first priority was to choose a source code management tool for all sites and to define a common methodology for usage of the tool. A working group consisting of members from each site assessed best-in-class configuration management tools. After an evaluation and pilot usage, we agreed to purchase Rational's Clearcase *. Within several months, the tool and a common usage methodology were deployed to all the developers in the department.

The methodology defines everything from directory

[1] Since a chip is very complex, it is natural to decompose it into smaller components. Each component is successively decomposed into smaller, more manageable components, thus creating a *hierarchy* of components. The connections between these components are referred to as design *connectivity*.

* All other trademarks are the property of their respective owners.

structure for the code and libraries to tagging of versions and naming conventions for branches. The result is that all Nike developers are now using the identical setup so that developers from one group can easily navigate the source code from another group. In addition to the above features, when using the configuration management system's branch-and-merge capability, developers can create and maintain multiple variants of their software concurrently. This allows them to easily move from the latest nightly build to a previous release and vice-versa.

In order to facilitate a high level of code sharing across the sites, we implemented an additional component of the configuration management tool that enables cross-site sharing of code repositories, namely Rational's Multisite*. After a developer has revised code and checked it in at one site, these changes are synched up at the remote site at intervals of 15 minutes. The replication is transparent to the user. This means that developers can access all modules in real time, whether they are being developed locally or remotely. A developer in Haifa cannot tell the difference between a module developed in Haifa and one developed in Santa Clara. Developers can share modules on the spot and debug integration problems over the phone. They can iterate this process easily and efficiently, as if they are sitting in the same building. The capability of instant code sharing is key to the tight integration between our performance verification tools and physical design tools, and to enabling the module sharing and reuse detailed earlier.

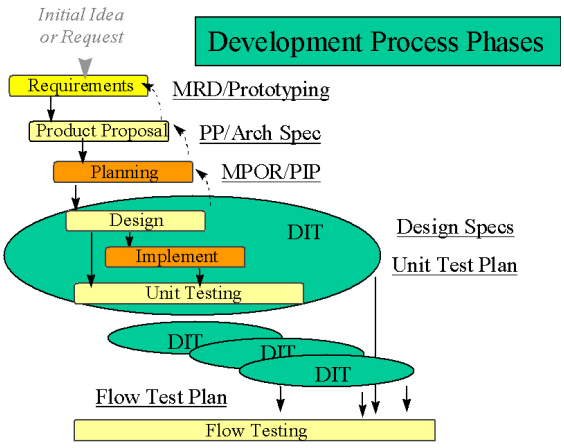

Figure 7: Software development cycle

Software Development Cycle

Another important component of the software infrastructure is the software development cycle. As shown in Figure 7, we defined an iterative software development process based on Design, Iterate, and Test (DIT) cycles. Each project breaks their development into small tasks whose DIT cycle is supposed to last no longer than 12 weeks. DIT cycles are designed to enable frequent synchronization points for validation. Once a quarter, we have a synchronization point where all the tools and modules synchronize to the same version of all libraries across all sites. This is critical in order to enable the modular architecture to work. The component sharing creates a large number of dependencies that we need to manage; the frequent synchronization points minimize the number of versions that need to be supported for each library.

As the common modules mature and stabilize, we have detailed a plan to move to weekly synchronization of the Nike libraries and eventually daily. Initially, there were too many frequent changes to the interfaces of the common modules to sync up more than once a quarter. In order to monitor the stability of the modules, we are tracking defects in the code through a bug-tracking system. After each synchronization point, bugs found in the libraries are reported internally and tracked in this system.

As part of the design phase, we require all Nike tools to write a Market Requirements Document, Product Proposal, External Product Spec, Internal Product Spec (when relevant) and a Test Plan. Each document in the design phase is approved by key customers, the system architects, and the software architects. Templates for all the documents are available for developers to help guide them through the process. In addition, we view prototype development as an important tool for gathering requirements and customer feedback and assessing new technologies. Prototyping may take place at any stage in the development cycle.

In order to achieve high quality in the code and design, we have implemented design reviews and code inspections across the organization. All critical code must be inspected, and authors are required to address major defects before an inspection is closed.

Validation

In the area of software testing, Nike is going for a breakthrough in both quality and productivity. After evaluating several test management systems from external vendors, we decided they did not meet our specific needs. Instead we chose to develop and deploy Olympus, an internally developed test management tool. Currently the first phase of development on Olympus is complete, and we are piloting the tool in each site. For improved productivity, Olympus pre-

sents the developer with a sophisticated front end through which he or she can complete test writing, regression building, golden results updating, and output analysis. Olympus integrates a user-friendly front end through which the developer creates and runs tests and regression on top of a back end that stores test data in an SQL database. Since all test data is centrally located in a database, multiple groups, across both sites, can share test data. Additionally, the details of the test are stored in the database and can be used for test planning even before the tests are completed. The database also adds an important dimension to our software development environment. Nike's testing is observable. Program management can query the database and find out how many tests have been created in the past month, how many have passed successfully, and how many are still not ready for regression.

Conclusion

In Nike, a major thrust of our development has been centered on defining a robust and modular architecture. The architecture is still evolving as development progresses. Our challenge is to sustain the architectural principles over the lifetime of the tool suite.

To date, we have invested significant effort in the definition phases of the data model and the infrastructure. We have observed that there is an overhead associated with software specification and definition due to the complexity of module sharing. However, we have also seen a significant reduction in the cost of implementation and integration. We perceive that this benefit will be even greater in the future once the foundation is complete and fully deployed.

Along with the benefits of a common data model, there are some associated risks. If the data model is not well managed, there is a potential for the data size to grow too large and complex. There is also a risk of organizational boundaries inhibiting development of shared modules if the sharing is not encouraged by management.

The changing design paradigm and constant evolution of technology brings forth a new set of challenges. A data model driven architecture takes us a step closer to a utopian "integrated, interactive, and incremental" system.

Acknowledgments

We thank the members of the Nike Product Engineering teams in Santa Clara and Haifa who have designed and implemented the software infrastructure as described here: Mark Ball, Shiri Cohen, Nina Galperovich, Miriam Kreisler, Ed Langlois, Neela Majumder, and John Ramirez. We also thank members of Nike Globals and Nike FCDE teams who were an integral part of defining and implementing the Nike data model architecture: Sridhar Boinapally, Vanco Burzevski, Yi-Hung Chee, Gil Kleinfeld, Zoya Korovin, Ronen Moldovan, Ran Ron, and Eric Tse. We thank the members of the Nike Architecture Board for playing a vital role in defining and consolidating the software architecture: Doug Braun, Yi-Hung Chee, Ganapathy Kumar, Mosur Mohan, and Siang-Chun The.

We thank the members of the Nike Product Engineering teams in Santa Clara and Haifa who have designed and implemented the software infrastructure as described here: Mark Ball, Shiri Cohen, Nina Galperovich, Miriam Kreisler, Ed Langlois, Neela Majumder, and John Ramirez. We also thank members of Nike Globals and Nike FCDE teams who were an integral part of defining and implementing the Nike data model architecture: Sridhar Boinapally, Vanco Burzevski, Yi-Hung Chee, Gil Kleinfeld, Zoya Korovin, Ronen Moldovan, Ran Ron, and Eric Tse. We thank the members of the Nike Architecture Board for playing a vital role in defining and consolidating the software architecture: Doug Braun, Yi-Hung Chee, Ganapathy Kumar, Mosur Mohan, and Siang-Chun The.

References

[1] Semiconductor Industry Association, *National Technology Roadmap for Semiconductors*, 1997.

Authors' Biographies

Veerapaneni Nagbhushan is currently a Nike software architect. Prior to that, he worked on several CAD tools at Intel. He holds a BE in electrical engineering from Birla Institute of Technology and Science and a M.S. in computer engineering from Syracuse University. Nagbhushan has been with Intel since 1987. His e-mail is vnagbhus@scdt.intel.com.

Yehuda Shiran is Nike Product Engineering Manager and Program Manager in Haifa. He holds a BSME and an MSME from the Technion in Haifa, a Ph.DME and an MSEE from Stanford, and an MBA from Haifa University. Yehuda has been with Intel since 1991. He previously worked in various Silicon Valley CAD development companies. His current interests include software development management infrastructure and software development discipline. His e-mail is yehuda.shiran@intel.com.

Satish Venkatesan is a senior CAD engineer in Santa Clara. He holds a doctorate in computer engineering from the University of Cincinnati and a BE in electrical engineering from the University of Roorkee. Satish has been with Intel since 1996. His e-mail is satish@scdt.intel.com

Tamar Yehoshua is currently managing the Nike Product Engineering team in Santa Clara and the PowerCAD team that provides CAD tools for low-power design. She holds a BA in applied mathematics from the University of Pennsylvania and an MS in computer science from the Hebrew University in Jerusalem. Tamar joined Intel's Design Technology group in 1993 where she has worked in CAD tool development and has held management roles. Prior to joining Intel, Tamar worked at the Institute for the Learning Sciences at Northwestern University. Her e-mail is tamar.yehoshua@intel.com.

Circuit Design Environment and Layout Planning

Bharat Krishna, NIKE-SC/Design Technology, Intel Corp.
Gil Kleinfeld, NIKE-HF/Design Technology, Intel Corp.

Index words: circuit design, layout planning

Abstract

Circuit design in deep sub-micron technologies requires that designers deal with numerous data, constraints, analysis, synthesis, and optimization tools. Although synthesis tools are widely used at Intel, new circuit technologies are evolving and are often not well supported by existing synthesis tools. Deep sub-micron technology requires that the impact of physical design be considered early in the circuit design phase to prevent costly circuit, layout, and sometimes, logic re-design. A common source of these re-designs is inaccurate assumptions about the layout aspects of the target design. Thus, early layout planning and accurate parasitics estimation must be done by circuit designers.

Intel's FUB Circuit Design Environment (FCDE) is an integrated, interactive, and incremental circuit design environment that incorporates multiple tools, data, constraints, and analysis tools. FCDE integrates all circuit design tools including circuit simulation, layout planning, parasitics estimation, timing analysis, circuit optimizers. Circuit design tools exchange data via a common data model (Unified Core Model). FCDE tools expose their functionality to each other by standard tool drivers that allow integration of in-house design tools as well as vendor design tools. FCDE was designed and implemented on the Windows NT* operating system, and uses native Windows* technologies such as Microsoft* Component Object Model (COM), Visual Basic* and Visual Basic for Applications* (VBA). This paper describes Intel's circuit design environment and its components, with special emphasis on the layout planner, and its role in circuit design flows.

Results from a recent microprocessor design project support the need for layout planning by showing that the amount of re-design and re-work required for blocks is reduced when early layout planning is carried out.

* All other trademarks are the property of their respective owners.

Introduction

Traditional custom integrated circuit design methodology can be depicted as a waterfall model, where a stage (e.g., logic design) is completed and the results passed on to the next stage (e.g., circuit design). Results at each stage are evaluated without any consideration of their effect on later stages. For example, interconnect loading is assumed during circuit design for technology mapping. This assumption is likely to be invalidated during the layout design phase. Thus, the final layout is analyzed to verify functionality. If there are problems, then the layout is re-designed in an attempt to correct them. If the redesigned layout fails to correct the problems, then a re-design at one of the previous stages in the flow is attempted. This flow (shown in Figure 1) is very time consuming and expensive due to the following reasons.

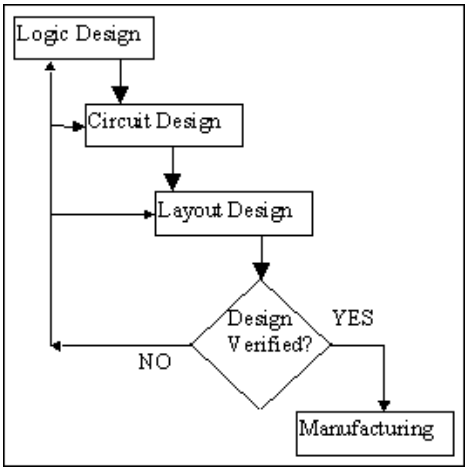

Figure 1: Traditional custom IC design flow

- *Many file formats*. The design flow uses many EDA tools. The tools may come from different companies. Even though some standard file formats (such as GDSII) have been accepted, there

are still many different file formats that contain data at different stages and many converters are needed to enable a complete tool suite. The existence of files as a means of data exchange also inhibits interactive design. The hundreds of files generated cause data storage problems, and maintaining data consistency during the design cycle becomes an issue.

- *Unnecessary task partitioning*. Different tools have inconsistent user interfaces and may be used by different engineers. This requires that the flow be partitioned between many engineers, each of whom focuses on local optimizations. Local optimizations usually inhibit overall design optimization as changes involve many engineers and can take several days.
- *Forces early binding*. Early bindings are decisions that are made at one stage that become hard inputs to the next stage. It is very difficult to change these input constraints later on in the flow, and they therefore inhibit optimizations at later stages.
- *Long iterations*. Since many tools are involved in a typical design flow, it takes too much time to iterate a flow. Many different engineers would be required to run their tools in the required order. This latency also inhibits the engineers' productivity. A design is reloaded many times into different tools, which significantly slows progress.

A New Methodology

A new design flow (shown in Figure 2) provides an interactive design environment. It enables overlapping of design stages so that the effects produced in later stages of the design can be easily considered in earlier stages. The overlap is enabled by providing easy access to multiple tools and by using a common data model across multiple domains. Since the system is developed as a native Windows NT* application, we make use of COM technology. All tools have ActiveX* interfaces such that one tool can invoke another tool. This is very much like the functionality available with many of the Windows tools where, for example, an Excel* spreadsheet can be created within a Word* document. This obviates the need for designs to exit a tool after saving its data, then to start another tool with previous data to analyze what-if scenarios. Since all the tools are required to understand a common data model, a tool can manipulate the design data, and any changes are available to another application to perform analysis or synthesis. This proposed methodology increases the complexity of tool development and necessitates the linking of internal data representations. However, the benefits of the new approach to the designer outweigh its cost by providing an instantaneous what-if glimpse across the traditional tool boundaries.

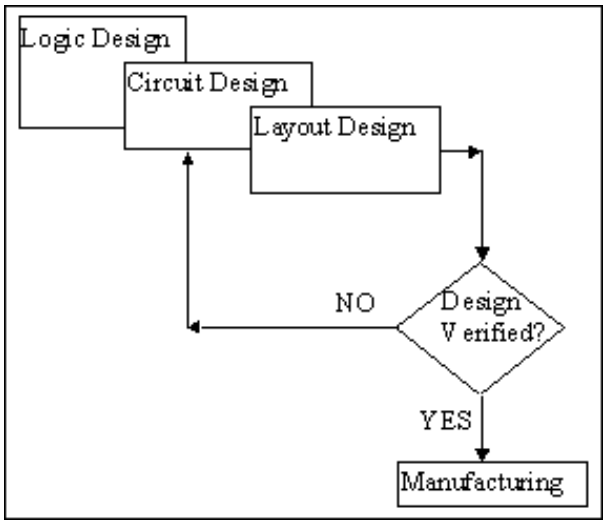

Figure 2: New design flow

The rest of this paper focuses on the interaction between the circuit design and layout design stages. We choose circuit and layout to illustrate the design stage overlap concept, and these can be similarly extended to other design domains. A workflow diagram illustrating the integrated design environment with these functionalities is shown in Figure 3. We describe in detail the functionality of the layout planner and show the advantages of the proposed system on the speed path optimization flow.

Layout Planner

Layout planning of datapath blocks has become very important due to the increasing complexity of the datapath and the tighter delay bounds imposed on critical signals. This is further emphasized by the increasing impact of interconnect delay on overall path delays. The layout planner provides functionality to estimate area and interconnect parasitics. This also allows the user to accomplish some layout tasks earlier, such as global routing, congestion analysis, track planning, etc., which later reduce the layout design effort significantly.

* All other trademarks are the property of their respective owners.

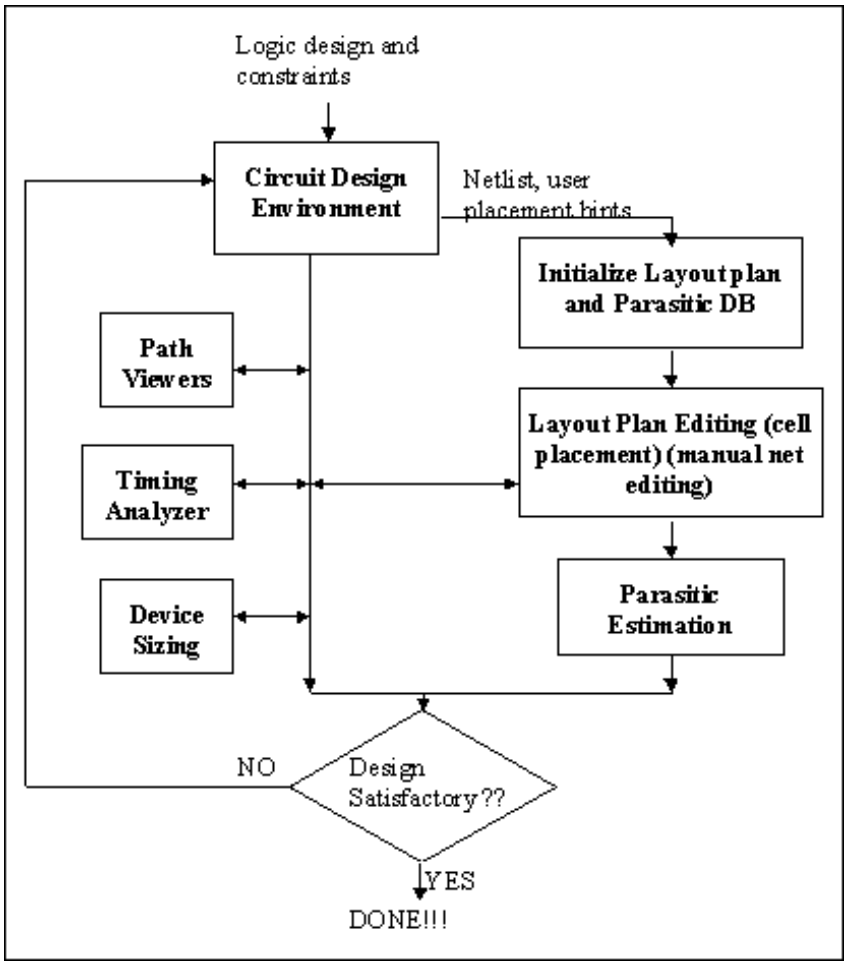

Figure 3: Workflow diagram circuit and layout interaction

The layout planner is a very important component of the circuit design environment. It has become necessary to incorporate accurate layout information (e.g., interconnect delays) during the circuit design stage in order to reduce design iterations. These iterations are necessary when the final layout does not satisfy all the assumptions made during the circuit design phase. Generally, circuit and layout design are done by different designers, and contemporary tools inadequately capture the design assumptions in the existing file formats. Layout planning of blocks is used to obtain early estimates for block area and timing of critical signals. The layout-based estimates are used during the circuit design stage to carry out more accurate circuit simulations and to design the datapath circuit schematics. The same layout plan is later used to drive the layout synthesis process. The layout planning methodology is designed with the following goals:

- be fast and highly interactive
- provide reasonable estimates for area and interconnects
- drive layout synthesis with a place and route plan
- enable what-if analysis and provide tradeoffs between accuracy and tool performance

The layout planning flow is developed from experience with prototypes used in some recent microprocessor design projects. Early layout planning provides a user with a means to estimate the layout area of a datapath block and the associated interconnect parasitics, which are used to perform quick performance verification analysis. The information obtained is then used to complete the design. The layout planning flow provides various trade-off points so that the circuit designer can get better estimates on interconnect design at the cost of tool performance. The user

can estimate interconnect parasitics derived from minimum spanning trees for rough estimates and actual global routes for more accurate estimates.

Functionality

The inputs to the layout planning tool are design constraints, user-defined placement hints, and the netlist (which may be incomplete). The tool provides a means to visually see and edit the placement and change the netlist. The netlist is modified if a cell used in the block is changed because of higher drive strength requirements, or any other interconnect optimization need. The key functions performed in the layout planning stage for interconnect optimization are as follows:

- cell area estimation and interface design
- placement of cells (as part of vectors)
- identification of vectors and rows
- global routing and congestion analysis
- parasitic estimation and timing analysis

Placement Modeling

A key feature of the datapath layout planning is the type of layout modeling that is used. Due to multiple instantiations of logic cells, common in datapath blocks, layout editing provides a means to edit groups of instances in one command. The multiple instances of a cell are grouped into entities known as vectors, the contents of a stage in circuit design are placed in a row, and the contents of a bit-slice form a column. Thus, the complete layout plan is modeled as a matrix. The tool then provides commands to move, delete, create vectors, rows, matrices, etc. This kind of modeling aids in ensuring regularity in the placement of cells in the layout plan.

Interconnect Estimation and Optimization

During layout planning the design engineers need to estimate the interconnect delays so that better data can be input to the circuit simulation stage. The interconnect length can be estimated by generating the Steiner tree [1, 2]. This estimation generates optimistic net lengths, as trees are generated for one net at time, and no consideration is given to obstructions or congestion due to other nets. For this reason, net length estimation also has a quick runtime.

Better net length estimates are generated by doing global routing. Global routing accounts for obstructions as well as congestion. It also considers physical net specifications (width, spacing). This option for net length estimation is slower than the Steiner estimation. For clock and other critical nets, some other tree estimation algorithm such as A-Tree [3] may also be used.

The layout planning tool provides these choices as runtime configurable user options.

Track Share Analysis

After a reasonable placement has been determined, the user is able to estimate interconnect parasitics. The location of the interface ports of the cell can also be planned to enable better routing [4, 5]. A typical routing, shown in Figure 4a can be improved with cell interface ports planning as shown in Figure 4b. The interface planning can be carried out using Track Share Analysis (TSA) or global routing. Based on the results of global routing or TSA, the interface terminals of the cells are placed at appropriate locations. The net length estimation process was successfully used in a recent project. By providing additional planning capabilities, we have given the designer full control of top-down as well as bottom-up aspects of datapath block layout design.

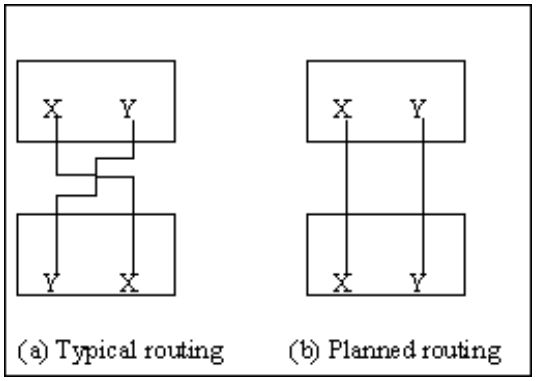

Figure 4: Cell interface planning

Visualization

The interactive graphical user environment provides other features. The user can plan for routing space and analyze routing congestion information that is derived from global routing. Based on the congestion analysis, the user can manually adjust the placement and plan out for area. The tool provides net visualization and editing functionality to interactively optimize the interconnect delay.

FCDE provides path viewing and debugging capabilities and includes the following viewers:

1. Paths list viewer. This is a list of all paths, including path properties (start point, end-point, margin, and more). The path list has links to a detailed path viewer, a schematic editor, and a layout planner.

2. Path detailed viewer/tracer. This viewer serves as the main debugging aid for speed-path analysis and optimization.

Incremental Design

The layout planning stage involves a lot of what-if analysis work. For example, the user may want to change the placement of a few instances and see how the area or timing on those affected nets has changed. To aid the user, the layout planner provides for incremental design. In this mode, the tool detects what nets have changed, and re-computes the desired properties for the affected nets only. The tool is also able to display the delta changes (difference between the property's old value and the new value). This allows the user to see immediately if the changes made are having positive or negative effects. Another advantage of incremental design is the efficient runtime, which is very important for an interactive design tool.

Cell Area Estimation

As layout planning happens before any real layout is created, an important requirement of the layout planner is to be able to estimate cell area from the netlist of the cell. The cell area estimator can provide various choices for estimation. We have estimated cell area in two ways and are experimenting with others.

The first method is to use a statistically derived equation for estimating cell area. The function's parameters include the number of devices, number of p-devices, number of n-devices, and the number of I/O ports. The second method is to use historical data. In this method, a table is created for various common types of cells, and the area of a cell is obtained by matching it against an entry in the table.

Other methods that we are experimenting with include the modification of the core engine of a cell synthesis tool. This method is expected to provide the most accurate estimates, but it is also expected to have the longest runtime.

Parasitic Estimation

Interconnect parasitic (resistance and capacitance) estimation is required since this information translates directly into delay information that can be used during circuit simulation. The parasitic estimation tool is designed to provide various run-time configurable options. These options allow the user to make tradeoffs between runtime and accuracy of the estimates.

Timing Analysis

In order to enable interactive design, a quick timing analysis engine is integrated into the layout planner. The quick engine yields rough timing analysis, but provides reasonable information on changes that occur when the layout plan is modified. Since the timing analysis data (slopes, timing widows, etc.) are shared between many circuit design tools (e.g., driver sizing), the timing analysis tool is a separate component of the circuit design environment. The layout planner invokes this engine when required, and the process of transferring data is transparent to the user.

Noise Analysis

Noise on interconnect is becoming a more visible problem with deep sub-micron designs. At present, the layout planner provides a means to visualize the noise as aggressor-victim pairs. This helps the user plan appropriate spacing between the aggressor and the victim. Current experimental work is directed towards auto identification of aggressors and victims. Based on the net topology, timing vectors on adjoining nets, and estimated cross-coupling capacitors, the proposed tool will be able to compute if there are any signal integrity issues, and then designers can either manually, or with the help of an automated router, reduce the net cross-coupling.

Device Size Tuning

FCDE will provide tight bi-directional communication with the schematic editor, a change notification system, and incremental capabilities. Using these capabilities, it will enable circuit designers to change the size of a device or a cell in the schematic editor or in one of the FCDE viewers. The change will be applied on the FCDE data model, and incremental analysis will be made to analyze this change.

New Functionality Enabled

Speed-Path Design

We describe the design environment and the advantages of using it by looking at the speed-path design activity commonly performed in IC design. This design activity encompasses both circuit design and layout design. It uses several tools and various types of data in both of the design domains. The flow of this activity is shown in Figure 5.

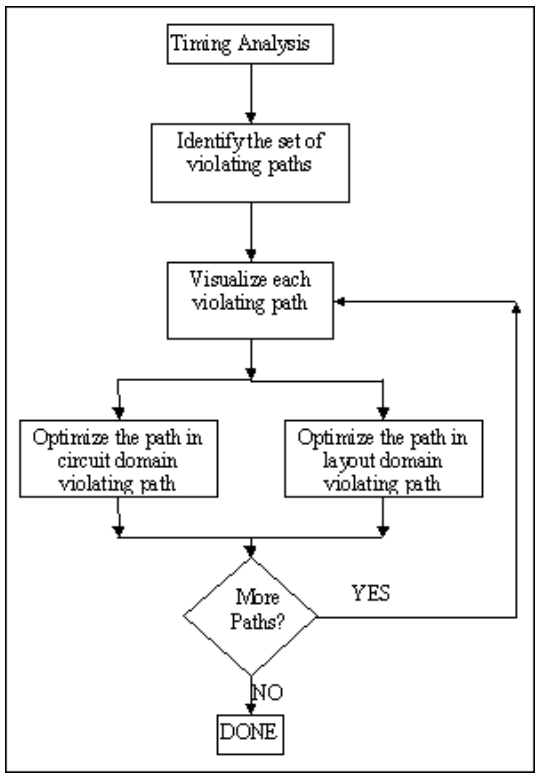

Figure 5: Speed path optimization flow

Speed-path analysis and optimization flow start with the timing verification stage when a circuit designer receives timing constraints and budgeting. The designer runs tools to identify critical paths. To optimize the paths with violations, there are multiple possibilities:

1. Increase the driver strength.
2. Reduce the driver load.
3. Optimize the interconnect loading.

The first two options are handled by the device-sizing functionality in circuit design, and the last one is carried out in layout design.

With current tools, all these activities (shown in Figure 5) are carried out by different tools. The data is exchanged by means of files and often there are multiple design engineers involved. Since data files are used, it is difficult to exchange partial data, i.e., complete design data is exchanged between the tools. By some estimates, it takes in the order of weeks to optimize a path if accurate interconnect loading is to be obtained. The reason for this is that layout design is carried out by a different person and the turn-around time for obtaining data is long. The most common reasons for the long turn-around time are that layout design has to be completed before data is obtained, and data interfacing is difficult. Thus, circuit designers tend to optimize the paths using device sizing and do not explore all possible solutions, such as optimizing the interconnect delay.

With the new integrated design environment, where all the different tools are accessible via a common user interface, the interconnects can be optimized as easily as devices can be sized. Also, since all the tools are working with the same data model, the data is exchanged in memory. This enables interactive design, which provides an improved turn-around time between various tools. Another advantage of the integrated design environment is that it provides the capability for incremental design. Only data that is modified by one tool needs to be addressed by other affected tools. With the integrated layout planner, a circuit designer is able to make changes to the block layout plan without involving a different person to do the layout design. Changes in interconnect parasitic values are updated in the common data model, and the timing analyzer is able to perform incremental analysis of the change.

Noise Handling

Another activity the new environment enables is the efficient handling of noise. It has become important to account for noise as the operating voltage for deep sub-micron design is decreasing, and the noise effect is becoming more visible. Noise analysis also involves information that is traditionally spread across both the circuit design and layout design stages. As both of these stages are tightly integrated in the new design flow, all the information required for noise analysis is available simultaneously.

Some of the noise analysis features made available have been adapted from published work [6].

Results

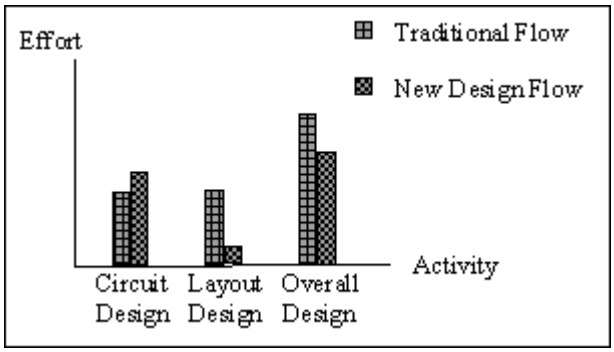

Figure 6: Gains in design time

Our results are illustrated in the bar chart shown in Figure 6. With the traditional approach, it is observed that the effort spent in circuit and layout design was almost equal. It is clear that when layout planning data are available during the circuit design stage, the circuit design time increases (about 1.25X). However, there is a significant decrease in the layout design time, which includes the time spent in fixing layout (about 1/3X). This reduces the overall design time by about 25%.

For a typical datapath block, the average time for post-layout fixing of critical paths improved from over two weeks to less than a week when the prototype of the proposed integrated design environment was used.

Future Work

Based on the layout plan, top-down cell templates are generated that need to be synthesized. During circuit design, these cells are placed and routed as per the interconnect plans. However, if a cell cannot be synthesized using top-down planning, a bottom-up correction to the overall block plan is generated. If several such iterations occur, a significant productivity penalty will be incurred. New cell synthesis algorithms are needed that can handle the top-down topology plans. Another area of exploration is to combine layout information even earlier in the design cycle, namely during logic synthesis. This will enable a designer to appropriately choose static or domino technology to meet the timing or area constraints and also insert the right number of pipeline stages during datapath design. All these call for a modification of traditional logic synthesis algorithms to account for layout effects.

Conclusion

By integrating circuit design with layout planning, we have improved the overall design time. Using early layout planning, we are able to incorporate accurate interconnect parameters into circuit design. This generates better timing analysis results, which match the actual post-layout timing analysis, thus reducing the need for re-design and re-layout.

Results from a recent microprocessor design project support the need for layout planning by showing that the amount of re-design and re-work required is reduced for blocks when early layout planning is carried out. In the speed path flow, expert design engineers have observed a three to five times improvement rate in the time it takes to fix violating paths.

Acknowledgments

We especially acknowledge Naresh Sehgal for valuable discussion and comments on the paper. We thank Ehud Kedar (former FCDE group leader), Gil Amid (Nike Timing analysis and circuit simulation group leader), Eitan Zahavi (Nike system architect), and Paul Madland (Intel Fellow, PMD circuit technology group) who have all contributed significantly to FCDE architecture and vision. We also thank Anurag Gupta, V Nagbhusan, and Manoj Gunwani from the Santa Clara NIKE departement, and Anat Ben-Artzi, Ronan Moldovan, and Arkady Neyshtadt from the Haifa NIKE department. We also acknowledge various members of other NIKE groups and the design teams who contributed to the realization of this methodology.

References

[1] M. R. Garry and D.S. Johnson, "The Rectilinear Steiner Tree Problem is NP-complete, " *SIAM J. Appl. Math*., Vol. 32, No. 4, pp. 826-834, 1977.

[2] J.P. Cohoon, D.S. Richards, and J.S. Salowe, "A Linear-time Steiner tree routing Algorithm for Terminals on the Boundary of a Rectangle, " *Digest of Technical Papers, ICCAD-88*, pp. 402-405, Nov. 1988.

[3] J.J. Cong, K.S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed RC delay Model, " *Proceedings ACM/IEEE Design Automation Conference*, 1993, pp. 606-611.

[4] B. Krishna, C.Y.R. Chen, and N. Sehgal, "Technique for Planning of Terminal Locations of Leaf Cells in Cell-Based Design," *Proceedings of 11th International Conference On VLSI Design*, pp. 53-58, 1998.

[5] Amnon Baron Cohen and Michael Shechory, "Track Assignment in the Pathway Datapath Layout Assembler," *Digest of Technical Papers 1991 IEEE International Conference on Computer-Aided Design*, pp. 102-105, 1991.

[6] D.A. Kirkpatrick and A.L. Sangiovanni-Vencentelli, "Techniques for Crosstalk Avoidance in Physical Design of High Performance Digital Systems," *Digest of Technical Papers, ICCAD-94*, pp. 616-619, November 1994.

Authors' Biographies

Bharat Krishna is the layout planner project leader in the NIKE/DT department. He received a M.S. degree in computer engineering from Syracuse University in 1994 and a B.Sc. degree in electrical engineer-

ing from the University of Khartoum, Sudan in 1991. He has worked for Intel since 1995 in the datapath layout automation area. His interests include datapath layout automation, VLSI routing, and physical CAD tool design. His e-mail is bharat.krishna@intel.com.

Gil Kleinfeld is the FCDE group leader in the Nike/DT department. He received a B.Sc. degree from Tel-Aviv University in mathematics and computer science. Gil has been working for Intel since 1988 in the areas of logic synthesis and datapath automation. Gil's main interests are automation of tedious design and verification tasks, and software design.
His e-mail is gil.kleinfeld@intel.com.

Challenges of CAD Development for Datapath Design

Tim Chan, Design Technology, Intel Corp.
Amit Chowdhary, Design Technology, Intel Corp.
Bharat Krishna, Design Technology, Intel Corp.
Artour Levin, Design Technology, Intel Corp.
Gary Meeker, Design Technology, Intel Corp.
Naresh Sehgal, Design Technology, Intel Corp.

Index words: datapath, synthesis, automation, and generation

Abstract

In many high-performance VLSI designs, including all recent Intel® microprocessors, datapath is implemented in a bit-sliced structure to simultaneously manipulate multiple bits of data. The circuit and layout of such structures are largely kept the same for each bit-slice to achieve maximal performance, higher designer productivity, and better layout density. There are very few tools available to automate the design of a general datapath structure, most of which is done manually. Datapath design (from RTL to layout) very often takes a significant amount of human resources in a project. The design is becoming more complex and demanding as the clock frequency is reaching 1GHz, and the process technology is getting to 0.15um and below. Issues with signal integrity, as well as leakage current, are much more significant now as VCC and VT continue to be reduced and current density increases. Elaborate analyses on noise and power are needed for future designs, beyond the already complex timing, reliability, and functional correctness analysis tasks. The burden on CAD tools to support the high-performance microprocessor design is bigger than ever. This paper reviews the general approaches used in the industry to design datapaths from RTL to layout with the difficulties and issues encountered. We propose a new design workflow and a set of tools to improve overall designer productivity, while meeting all other constraints. A description of these tools to support the next generation of microprocessor design is also presented. Our proposed flow allows a designer to choose a design methodology ranging from a fully automated one to a custom one, to a flexible mix of the two. We present a new paradigm of early binding that considers the impact of circuit and layout during RTL design. We also strive to preserve RTL regularity during the circuit and layout design to improve time-to-market. Finally, we present some results on actual design blocks with the proposed tools and workflow, and we suggest future areas for further research.

I. Introduction

In most microprocessor design projects, the design team includes computer architects (particularly important for a design with new architecture), micro-architects (who determine the amount of hardware resources to be put in the chip and how the major data flow occurs), logic designers, circuit designers, and layout designers. Sometimes, these designers may have overlapping functions (for example, doing both circuit and layout design) depending on the experience level of the designers and the project management philosophy. Nevertheless, in general, designers of different disciplines need to communicate at different levels of design abstraction, and a design can only be completed when design data at different abstraction levels are consistent with each other and correct, meeting the design objectives.

1.1. Traditional Datapath Design Flow

For most high-performance microprocessors, the workflow for datapath design involves many labor-intensive steps [1, 2]. Logic designers and micro-architects determine the detailed features of hardware and the methods used to achieve particular functions. The number of pipe stages and which operations go with each pipe stage are also determined. These decisions are made with the help of bottom-up circuit

feasibility studies and some estimation tools for timing and area. The processes of developing the most appropriate computer architecture, micro-architecture, or RTL are also very involved, but they are beyond the scope of this paper. The starting point of the workflow is a partition of RTL coding for which timing and area estimations have been made and the results are within acceptance tolerance.

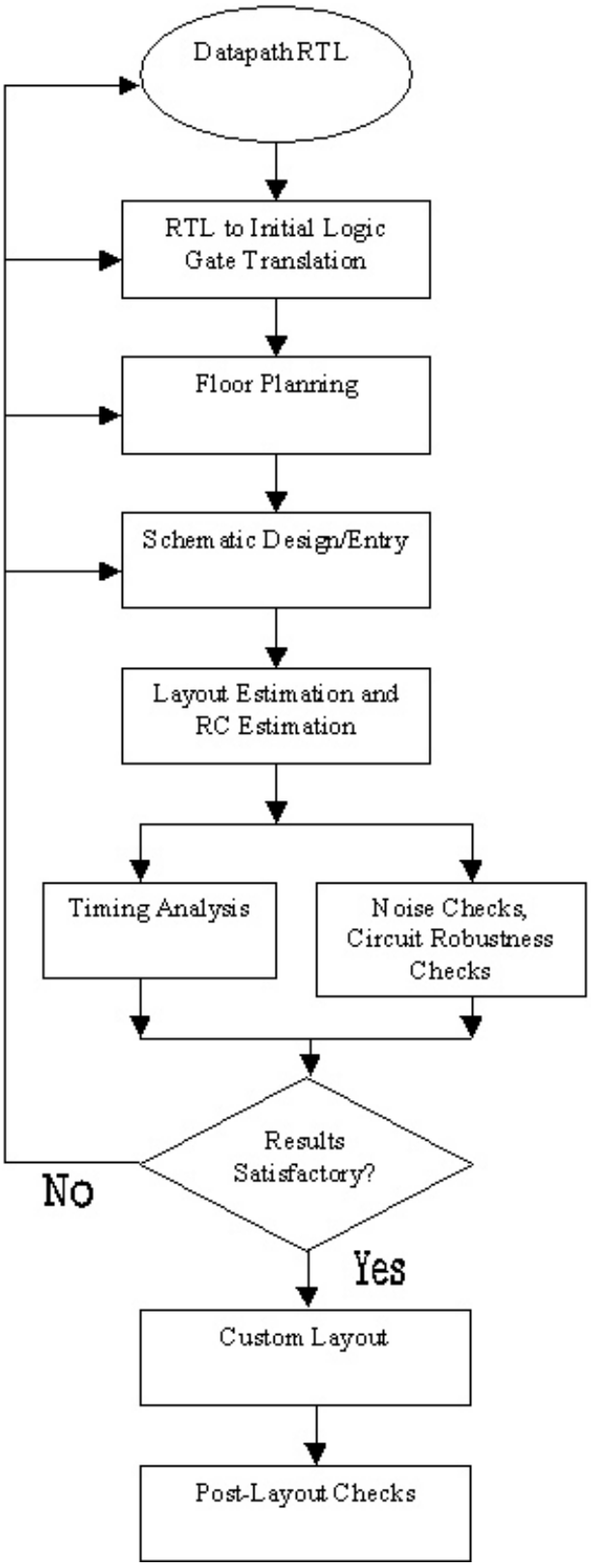

Figure 1: Datapath design flow

1.1.1. RTL to Initial Circuit Translation and Floor Planning

A circuit designer needs to study the functionality of the block first and then come up with the initial implementation plan that includes logic stages (without complete transistor sizing), floor planning, and circuit styles, based on different timing, area, and routing constraints. Depending on how experienced the designer is with the design techniques required for the block, feasibility studies and evaluations of a few design options are generally required to arrive at an implementation plan.

1.1.2. Schematic Design and Interconnect Estimation

The process of schematic design and interconnect estimation follows. As interconnect capacitance can be a significant portion of the total capacitance of a signal, a designer has to estimate the interconnects based on the assumptions made in his/her floor plan. In order to get a reasonable floor plan before actual layout is available, layout cell areas and pin/port locations are estimated. The "estimated" layout cells will be tiled according to the floor plan. With this rough datapath cell placement completed, interconnect lengths can be estimated based on the cell locations.

Circuit topology, transistor sizes, and floor plans will continue to evolve until a satisfactory design implementation is reached (though sometimes, the design specifications and external interfaces might also need to be modified). Once the schematic database for the datapath block is established, many checks and analysis can be performed.

1.1.3. Design Analysis

Next, the design with transistor sizes and interconnect RC's can be analyzed quite accurately for timing and many other circuit robustness requirements such as race conditions, noise tolerance, and long-term device and interconnect reliability. If the results of the analysis are not acceptable, the routing, cell placement, schematic design, RTL design or a combination of these will need to be modified.

1.1.4. Custom Layout and Post-Layout Checks

When the schematic with the corresponding estimated layout is satisfactory, layout can then be custom designed. Incorrect assumptions used in the estimated layout are corrected, and manual placement optimization is used.

When actual layout is completed, RC extraction is performed, and the RC netlists are merged back into the schematic netlist. At this point, all analyses of timing, noise, and circuit robustness are performed on the "accurate" netlist to verify that the design with actual layout data still meets the design requirements. If there is any problem found with the design, the design process is iterated until the design requirements are met.

With the top-down design process, even though the design requirements at one point are met, since other blocks in the chip might require design changes, the block needs to go through the Engineering Change Order (ECO) process. This is a formal procedure to communicate and implement changes to meet new requirements. In other words, the design process is reentered.

1.2. Issues with Traditional Datapath Design Flow

This design process mentioned above is quite top-down driven and sequential. From RTL to circuit and then to layout, each step makes a set of assumptions/estimations and provides more accurate information than the previous step. Each step of the design process also takes a significant amount of time to finish. As a result, poor estimations in early steps have very costly consequences due to the amount of time and effort required to make changes. In a large design project with a large team, the problems get multiplied many times over when poor estimates from one team member affect the design of other team members. As design specs are changing, implementations are not stable. Both become moving targets, and communication overhead increases substantially. As we have observed, large projects tend to require a long time for design convergence (i.e., when different pieces fit together and meet project requirements), and they have lower design productivity.

1.3. The Direction of Higher Levels of Automation

The accuracy of early estimations, and the turnaround time for the major design steps (e.g., RTL to circuit design) are very important elements when considering productivity in the design process. (Company culture, team maturity, design and management experience level are also part of the puzzle; however, these are not discussed in this paper.) Interestingly, more accurate early estimates and faster turnaround time can both be achieved with design automation. Automation that provides the correct result quickly can shorten the turnaround time, and it can also give more accurate estimations for the options that designers want to explore by quickly implementing these options. Though it is by no means easy to automate the design for high-speed complex microprocessor design using deep sub-micron technology, a lot of effort has al-

ready been made in the academic arena and by EDA companies. Automatic datapath cell layout generation and datapath block place and route tools are now commercially available. Compilation to datapath circuit-level netlist from hardware description language is also getting popular (in the less performance-critical designs such as chip set design). Over time, manual circuit and layout design techniques are digested by CAD developers who then formulate methods and heuristics to solve these design problems with CAD tools. However, this work is just not done soon enough to alleviate the burden of the designers for high-performance designs.

1.4. Structure of the Paper

The objective of this paper is to share our view of the high-performance datapath design problems and our ideas of what the solutions will look like, by providing some details of our work. Naturally, we don't have all the answers, but we believe that we have some good ideas about how these problems should be approached. It is hoped that this paper will stimulate readers to come up with more and better ideas.

The next section describes a more automated workflow compared to the workflow just described. The new workflow features automatic schematic generation from RTL and layout synthesis. The techniques of regularity extraction and how they are used in logic synthesis and schematic generation are discussed in the section on datapath logic synthesis. An efficient approach to design datapath schematics and to layout planning together is described in the section on datapath layout planning and placement. Accurate and efficient RC estimation is essential to various steps of the design process, and it is discussed in the section on the parasitic estimator. Datapath cell layout synthesis significantly reduces layout design resources for high-performance design, and it is discussed in the section on layout cell generation.

II. A More Automated Design Flow

A new workflow, which drastically improves productivity, is shown in Figure 2. It supports synthesis from RTL to layout, though with the understanding that datapath synthesis techniques will take time to mature. Designers input and user interfaces are essential to every step of the process.

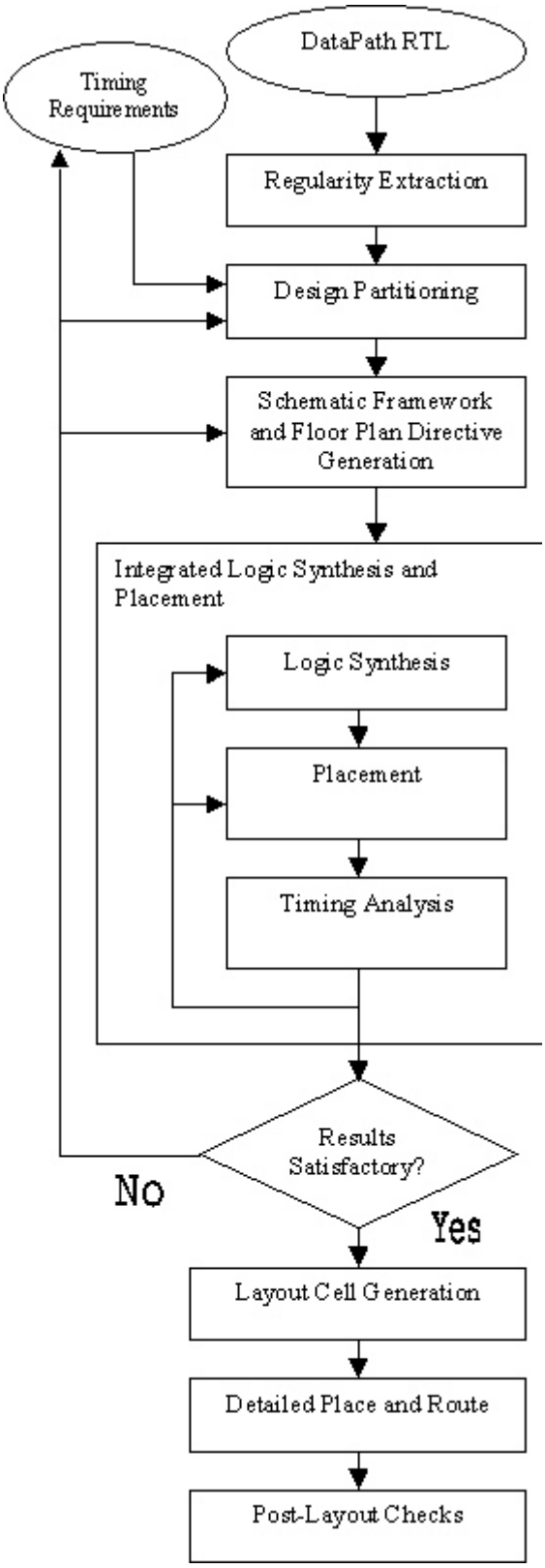

Figure 2: A more automated datapath design flow

The rest of this section describes the rationale behind each component of the design flow and gives the high-level expectations of these steps.

2.1. Regularity Extraction

In this workflow, regularity extraction is performed first to identify the repeated functionalities in RTL and to come up with optimal logic templates for logic synthesis at later stages. A good template needs to contain a few stages of logic (at least) to allow for the synthesis tool to perform optimization; however, it cannot be too large (containing too many logic functions) such that a lower level of regularity in circuit and layout cannot be exploited and therefore cause device density and performance to suffer as a result. In fact, the main reason for doing regularity extraction is the inability of the current synthesis tools to produce regular structures from RTLs for repeating functions. Secondly, with a logical netlist available from regularity extraction, designers can control the degree of regularity used in synthesis and modify the outcome of synthesis more easily.

2.2. Design Partitioning

This step is performed to identify what circuit style to use for different parts of the design. The commonly known circuit styles are static and dynamic. Normally, static is the first choice due to the robust nature of the style and the ease of design. However, in terms of speed, dynamic circuits are generally about 30% faster, and this style needs to be used when the speed of the circuit is critical. The price of using dynamic circuits is higher power consumption and greater design effort. As for high fanin logic, the use of dynamic circuits is more advantageous. Thus, a design partitioner is expected to estimate the timing performance of the datapath block with static circuits and single out the paths that are not meeting the performance requirements. Once some sections of the logic have been identified for dynamic circuit implementation, the logics going in as input to these dynamic circuits need to be considered as candidates for dynamic implementation as well, in order to ensure correct circuit functionality.

Also, it is expected that the current logic synthesis tools are not able to produce optimized results for complex special functions, such as a 32-bit adder (which involves a lot of special circuit techniques and fine-tuning). A datapath macro cell library (probably with special macro cell-sizing techniques) needs to be used to supplement the deficiency of current synthesis tools. As a result, the design partitioner needs to identify the logic functions that should be supported by a macro cell library (such as adders, register files, and comparators) and later target those functions for macro cell mapping and sizing.

2.3. Schematic Hierarchy and Floor Plan Directive Generation

Schematic hierarchy generation follows after design partitioning is done, and even though at this point no actual logic gate or transistors have been mapped, a schematic hierarchy with logic templates can be created. With schematics, circuit designers can proficiently modify the design partitioning and hierarchy for better synthesis results. Again, it is not expected that perfect results can be achieved by the design partitioner, and input from the designer is very crucial at this point. With regularity reflected in the hierarchical schematics, designers can modify the datapath cell placement directives (for placing cells into rows and bit columns) that are created by tools using heuristics.

2.4. Integrated Logic Synthesis and Placement System

Once the partitioner has been given input for synthesis and directives for placement, the integrated synthesis and placement phase is entered. The main reason an integrated system for synthesis and placement is needed is that doing logic synthesis without placement information does not give good enough results for future process technologies (0.15um or below). Transistor intrinsic delay continues to improve, and the average percentage of interconnect capacitance over the total node capacitance continues to increase. Interconnect delay has become an important component in very high-performance design, and the traditional wire load model used in control logic synthesis is not adequate for high-performance datapath synthesis. Placement information (in turn, RC information) needs to be available for the synthesis tool for correct sizing, buffering, signal repeating, and circuit topology choice.

2.5. Integrated Schematic Design and Layout Planning Environment

In the same spirit, designers need to be able to interact with schematics (outcome of synthesis), and placement needs to be integrated into the design tools. The tools have to efficiently support modifications of schematics and placement (RCs) by the designers, and be quickly able to communicate the changes among themselves to enable designers to see the effects of their changes (on timing, area, power, and noise, etc.).

2.6. Layout Cell Generation

When logic synthesis and global placement are completed, layout cells at the layout hierarchy assigned by the placement tool are then generated. A lot of meth-

odology definitions have to be completed before layout generation, such as power griding structures and usage of metal layers for cell pins and ports. Metal width and space requirements for reliability and noise concerns are also considered.

Layout cell generation is not the only way to create the bottom hierarchy of the layout. Library cells can also be used as in the traditional control logic layout synthesis. The layout quality of library cells is expected to improve as more effort has been put into library cells that are expected to be used by different projects. However, layout density might not be as good when compared to layout done with cell generation, since cell generation processes more devices together and has the opportunity to achieve better optimization.

2.7. Detailed Place and Route

After layout cells are generated, they can be used for detailed place and route (which is the process of generating DRC-clean placement and routing, based on the approximate (sometimes incomplete) results from global placement and routing. If global place and route are done well, it is expected that detailed place and route will only change the RC results by 5%. When a DRC-clean layout is completed, RC extraction can be performed, and all the necessary post-layout analysis can then be done with accurate RC information. The analyses normally include electrical rule checks, noise, timing violations, and setup and hold time checks (min-delay analysis).

Now that we have outlined the overall design flow, we focus on the details of the major design steps in the following sections.

III. Datapath Logic Synthesis

Logic synthesis, which transforms a design from RTL to circuit level, has been widely studied for control logic. Logic synthesis [5, 6] involves two steps: logic minimization followed by technology mapping to a user-specified library. Datapath circuits possess a very high degree of regularity that has to be preserved throughout the design process to achieve high density and performance. If the traditional logic synthesis approach based on logic minimization is used, then some regularity would be lost, resulting in inferior results. Therefore, an ideal datapath synthesis approach should first extract the regularity inherent in RTL descriptions prior to mapping the circuit to a desired technology. The extracted regularity results in a design hierarchy, which should be preserved to achieve high design quality as well as productivity.

We propose a novel methodology for logic synthesis of datapath circuits, where the datapath regularity is first extracted and then the circuit is mapped to a desired technology while preserving regularity. The input to our synthesis approach is an RTL description of a datapath circuit. Regularity in the circuit implies the existence of subcircuits, called templates, which have multiple instances in the circuit. Regularity extraction first identifies a sufficiently large set of templates and their instances, and then completely covers the circuit by a subset of these template instances. The template instances are then grouped to form datapath vectors. A schematic of the datapath is generated using these vectors and the boundary constraints on the I/O buses and signals. The schematic helps the designer in understanding the circuit and in making important decisions about or changes to the templates and vectors identified so far. The next step is to map the templates to static and dynamic logic as desired, thus resulting in efficient multi-technology designs. Finally, the mapped templates are sized according to the loading on the primary outputs of the circuit.

```
Module Example1

Inputs      a[3:0], b[3:0], c[1:0];
Inputs      sel1, sel2 sel3, sel4;
Outputs p[3:0];

begin main
    for i = 0 to 1 do
        f[i] = cond begin
                    [sel1]      b[i];
                    [sel2]      c[i];
                    []          0;
                    end;
    for i = 2 to 3 do
        f[i] = cond begin
                    [sel1]      b[i];
                    []          0;
                    end;

    /* increment f[3:0] to get g[3:0] */
    /* f[3:0] = g[3:0] + 1;                    */
    /* we use the following gate-level representation */
    /* of the incrementer for illustrating our approach */
    carry[1] = f[0] AND f[1];
    carry[2] = carry[1] AND f[2];
    carry[3] = carry[2] AND f[3];

    g[0] = NOT f[0];
    for i = 1 to 3 do
        g[i] = carry[i] XOR f[i];
    /* end incrementer*/

    for i = 0 to 3 do
        p[i] = cond begin
                    [sel3]      g[i];
                    [sel4]      a[i];
                    []          0;
                    end;
end main;
```

Figure 3: HDL description of a small datapath circuit used to illustrate our synthesis approach

We describe below in detail the various steps in our synthesis methodology of datapath circuits. We explain our methodology with the aid of the circuit in Figure 3 (the corresponding logic diagram is shown in Figure 4).

3.1. Regularity Extraction Techniques

The task of regularity extraction is to identify a set of templates and their instances from the RTL description of the circuit (*template generation step*), and then to cover the given circuit by a subset of these templates (*circuit covering step*), where the objective is to use large templates that have a large number of instances. Figure 5 illustrates a circuit cover with four templates, where template T1 has six instances, T2 has three instances, and so on. The extraction step involves a tradeoff, since a large template usually has a few instances, while a small template has a high number of instances. Note that the template composed of T2 followed by T1 has only three instances, compared to six instances of T1. Usually, a large template implies a better optimization of area and performance, while a template with more instances requires less design effort, assuming a template is synthesized only once for all its instances.

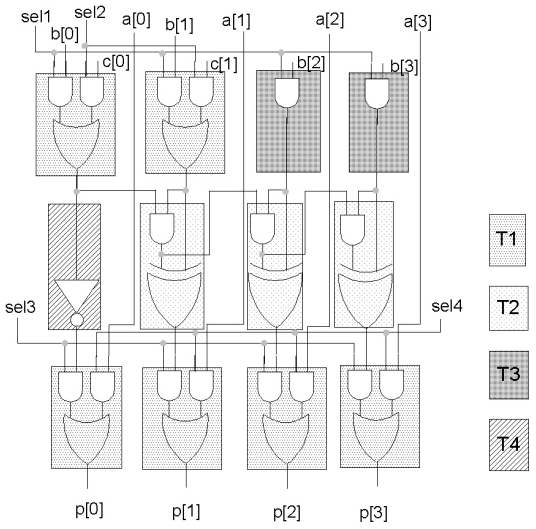

Figure 4: Logic diagram of the circuit of figure 3; the four templates shown here form a circuit cover

Several techniques for extraction of functional regularity have been proposed in the literature [3, 4, 8, 9, 10, 11, 12]. Most of these techniques focus on covering a circuit by templates, assuming that a library of templates is provided by the user. Very few techniques address the problem of generating a good set of templates. Given a library of templates, Corazao *et al.* [8, 11] address the problem of mapping a circuit described at a behavioral level using templates from the target library. Their approach addresses several key subproblems, such as finding complete as well as partial matches of a template and selecting a good set of templates to optimize the clock period. Rao and Kurdahi [12] represent the input circuit as well as templates from the given library by strings, and they use a string matching algorithm to find all instances of the template in the circuit. These authors present heuristics to generate a set of templates; the final cover is highly sensitive to these templates. Odawara *et al.* [9] present a methodology to identify structural regularity in highly regular datapaths. In their method, latches driven by the same control signals as initial templates are chosen and used to grow larger templates. Odawara's approach identifies one-dimensional regularity in terms of bit-slices of the datapath. Other approaches by Nijssen *et al*. [10] and Arikati *et al.* [3] extend Odawara's methodology to identify bit slices as well as stages of datapath circuits. These structural methods perform well for highly regular circuits, but might not work for circuits with a mix of datapath and control logic. A problem similar to regularity extraction is technology mapping, where the input circuit is covered by cells (templates) from a given library. Keutzer [7] proposed partitioning the circuit into rooted trees and then mapping the trees using library cells, by using dynamic programming. All the abovementioned techniques address the problem of covering a circuit by templates, where the templates are either provided by the user or generated in an ad hoc manner. None of these techniques deal with the systematic generation of a set of templates for a given circuit.

We have designed an efficient and robust approach for extraction of functional regularity [13, 14], where the set of all possible templates is generated automatically for the input circuit under two simplifying, yet practical assumptions: (a) only maximal templates are considered, where a template is maximal if and only if all its instances are *not* entirely covered by instances of another template, and (b) input permutations of gates in the RTL description are ignored. The number of templates is reduced to within V^2, where V is the number of components in the circuit. We have demonstrated that a wide range of efficient covers are obtained for various benchmarks from the set of templates generated by our approach [13]. Since a sufficiently large set of templates is generated, and the binate covering problem is inherently difficult [5], we employ simple and efficient heuristics to cover the circuit. Our approach recursively selects a template from the complete set of templates, based on one of the following heuristics, and deletes all its non-overlapping instances from the input circuit, until the entire circuit is covered.

(a) Largest-fit-first (LFF) heuristic: select the template with the maximum area, where the area of every component is given.

(b) Most-frequent-fit-first (MFF) heuristic: select the template with the maximum number of instances.

These two heuristics give different covers; other heuristics can be used to generate a range of covers from which the designer can choose the most desirable cover. In fact, we can represent the set of covers by a template hierarchy, where regularity among different templates is recursively extracted [14]. In the event that a template is specified by the designer, using our approach, all its instances can be generated and used for finding a cover. Thus, a cover, which is a mix of automatically extracted and user-specified templates, can be generated. (We have filed a patent on our regularity extraction approach [15]).

3.2. Vector Identification

So far, we have generated templates using functional regularity [13] without accounting for the circuit structure in terms of the interconnections among the template instances. As a result, the templates do not directly correlate with the datapath vectors. For example, the six instances of template T1 in Figure 4 should belong to two different vectors. (Here, a vector is defined as a set of template instances that are grouped together for subsequent synthesis and layout stages.) We now consider structural regularity to transform templates into datapath vectors [13]. We explain the steps of vector identification using the example of Figure 4; the resulting vectors are shown in Figure 5.

- *Simple vectors*: The instances of a template are partitioned into vectors, which we call simple vectors. For example, the template T1 of Figure 4 is partitioned into two simple vectors, SV1 with two instances and SV2 with four instances. The remaining templates result in a single simple vector each.
- *Composite vectors*: Simple vectors of different templates are grouped, if possible, to form composite vectors. For example, simple vector SV1 of template T1 is grouped with the simple vector of template T3 to form a composite vector V1 (see Figure 5).

The resulting vectors of the template cover of Figure 4 are shown in Figure 5. We use a set of efficient heuristics to group template instances to form simple or composite vectors. These heuristics are listed below.

1. *Control/data inputs*: The input signals of template instances are classified as control or data from the HDL description of Figure 3, e.g., *sel1* is a control signal, while *a[0]* is a data signal. The instances with the same control inputs and similar data inputs are grouped together.
2. *Output signal name*: The instances whose outputs drive the same bus are grouped together. For example, the instances of templates T2 and T4 are grouped together, since their outputs form the bus *g[3:0]*.
3. *Circuit topology*: Two template instances are grouped in the same vector, only if one of them is not in the transitive fanin of another. This heuristic will ensure that the template T1 (Figure 4) would be partitioned into at least two simple vectors, since two of its instances are in the transitive fanin of two other instances.

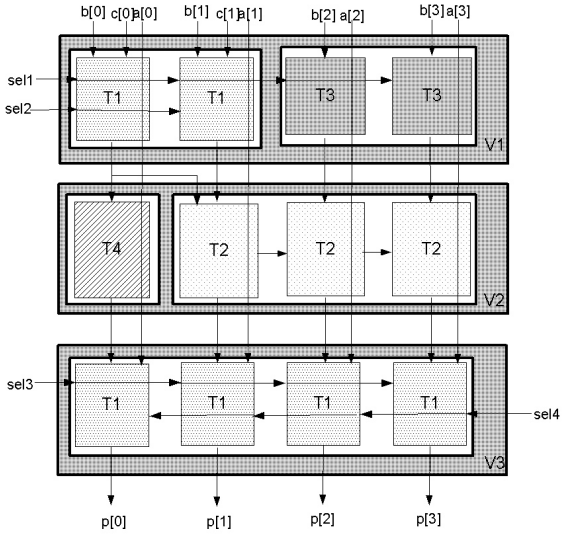

Figure 5: Schematic of example circuit obtained after forming vectors from the templates of Figure 4

3.3. Schematic Generation

A schematic of the datapath circuit is generated using the vectors identified earlier and the control/data assignment to the signals. The schematic for the example circuit is shown in Figure 5. The schematic is essential to allow designers to control the design process: (a) they can get a much better understanding of the circuit than they could from the HDL description; (b) they can modify the design hierarchy and floorplan by merging/breaking templates or vectors, changing the control/data orientation of signals, and modifying the order of vectors. An example of such a modification is merging templates T2 and T1 to form a larger template with three instances, which might lead to better optimization during subsequent steps.

3.4. Technology Mapping

The input to technology mapping is the set of datapath vectors and the I/O timing requirements in terms of input arrival times and output loads. The partition of

the circuit into vectors (or underlying templates) allows the designer to select a desired technology for each template independently. Currently, our synthesis flow assumes that the mapping of templates is performed manually, which can be easily automated due to the small size of templates. We explain several choices for mapping of templates.

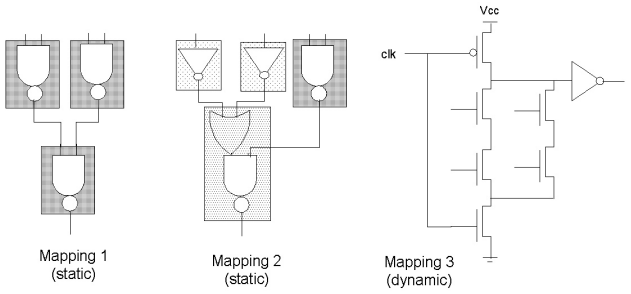

Figure 6: Several mappings of template T1 (Figure 4) to static and dynamic logic

- *Static logic*: The traditional approach of mapping a circuit to static logic first decomposes the circuit into smaller directed-acyclic graphs (DAGs) [5, 7] and then independently maps each DAG to the specified library of static cells. In our case, the templates are small enough to be mapped directly without any decomposition. Figure 6 illustrates two mappings of template T1 (Figure 4) to static logic. Here, mapping 2 is suitable for template T1 in vector V3 of Figure 5, since one of the data input signals arrives later than the other. On the other hand, mapping 1 is suitable for T1 in vector V1 (Figure 5). Thus, depending on the template usage in the circuit, we might have to use several mappings of a template.
- *Dynamic logic:* A template can be mapped to dynamic logic to achieve better timing; however, noise-related issues have to be considered, such as the length of the input and output signals of the template. Figure 4 also shows a mapping of template T1 (Figure 4) to dynamic logic. We are looking into automating the mapping of templates to dynamic logic.
- *Macro cells:* Datapath circuits employ commonly occurring logic blocks, such as incrementers, adders, shifters, etc. A library of various mappings of these specialized datapath blocks for a range of area, performance, and power values will be required. For example, the incrementer in the example circuit of Figures 3 - 5 can be replaced by one of its mapped versions prior to extracting regularity from the HDL description; our synthesis flow would then result in vectors V1 and V3 shown in Figure 5, while V2 would correspond to a macro cell.

3.5. Gate Sizing

Once all the templates of a circuit are mapped to the desired technology, every gate is sized to satisfy the output load requirements. The output load capacitance of a gate comprises the following components:

1. *Gate capacitance*: The capacitance values of the gates driven by this particular gate are available after the technology mapping step.
2. *Diffusion capacitance*: The diffusion capacitance of the gate is also known after technology mapping.
3. *Interconnect capacitance*: The capacitance is available only after the post-synthesis steps of floorplanning and RC estimation. Therefore, interconnect capacitance is used only in the gate-sizing step in the subsequent design iterations.
4. *Primary output load capacitance*: The load capacitance is already specified for the primary outputs of the circuit.

The sizing of the gates of the mapped circuit is performed starting from the primary outputs and traversing back to the primary inputs, where the output load requirement is satisfied for every gate encountered. If there are loops in the circuit, then the gate sizes will take a few iterations to converge.

Different instances of a template mapping will be sized differently depending on the output load requirements. In general, a template with multiple instances can have several mappings, where each mapping can have several different gate sizes.

Gate sizing is performed again after the interconnect capacitance values are obtained from the floorplanning and RC estimation steps.

3.6. Results

While the steps of technology mapping and sizing are still under development, we have implemented prototypes for regularity extraction and vector identification. We list below the results of regularity extraction and vector identification on two datapath blocks in terms of the number of templates, vectors, and their instances.

Ckt.	No. of components	No. of templates (instances)	Regularity index	No. of datapath vectors
Block1	464	9(400)	2%	7
Block2	318	17(124)	12%	9

We have defined an index, called a *regularity index*, to evaluate the results of regularity extraction [13]. The regularity index is defined as the percentage of the number of logic components in all the templates to the total number of logic components in the circuit. The regularity index correlates to the reduction in the design effort, assuming that a template is not synthesized multiple times for its multiple instances.

IV. Datapath Block Floorplanning and Placement

4.1. Objectives of Layout Planning

Layout planning of datapath blocks is used to obtain early estimates for block area and timing of critical signals. The layout-based estimates are used during the circuit design stage to carry out more accurate circuit simulations and design of datapath circuit schematics. Layout planning support should provide the following:

- speed and high interactivity (to enable what-if analysis)
- reasonable estimates for area and parasitics
- tradeoffs between accuracy and tool performance

We have developed a set of tools, based on experience from a recent microprocessor design project. These tools provide a user with the means to estimate the layout area of a datapath block and the interconnect parasitics from which quick timing analysis can be performed. The designers can also estimate interconnect parasitics derived from minimum spanning trees for rough estimates and actual global routes for more accurate estimates.

4.2. Tasks in Layout Planning

The inputs to the tools are top-down block pin interface, user-defined placement hints, and the netlist (which may be incomplete). The tool provides a means to visually see and edit the placement and change the netlist. The netlist is modified if a cell used in the block is changed because of the need for higher drive strength or other interconnect optimization requirements. The key functions performed in the layout planning stage for interconnect optimization are as follows:

- cell area estimation and interface design
- identification of vectors and rows
- placement of cell instances (as part of vectors)
- global routing and congestion analysis
- parasitic estimation and timing analysis

A key feature of the datapath layout planning is the layout modeling. Due to the frequent occurrence of multiple instantiations of a logic cell in datapath blocks, an entity called a vector is created to represent a group of instances, and layout editing on these groups of instances is supported. Further, it is also observed that the contents of a stage in the circuit design are placed in a row and that the contents of a bit-slice are placed in a column. Thus the complete layout plan is modeled as a matrix. Commands are then provided to move, delete, and/or create vectors, rows, matrices, etc. This method of layout modeling helps ensure regularity in the placement of cells in the layout plan.

After a reasonable placement has been determined, a designer will then estimate interconnect parasitics. Location of interface ports of the cell can also be planned to enable better routing [16, 17]. The interface planning can be carried out using Track Share Analysis (TSA) or global routing. Based on the results of the global routing or the TSA, the interface terminals (pins and ports) of the cells are placed at appropriate locations, and the net length estimation process proceeds.

An interactive graphical user environment has been developed to support this layout planning process. This environment also provides other features. A user can plan for routing space and analyze routing congestion information, which is derived from global routing. Based on the congestion analysis, the user can manually adjust the placement and plan out for area. The environment also provides net visualization and editing functionality to interactively optimize the interconnect delay. The overall design flow for layout planning is shown in Figure 7.

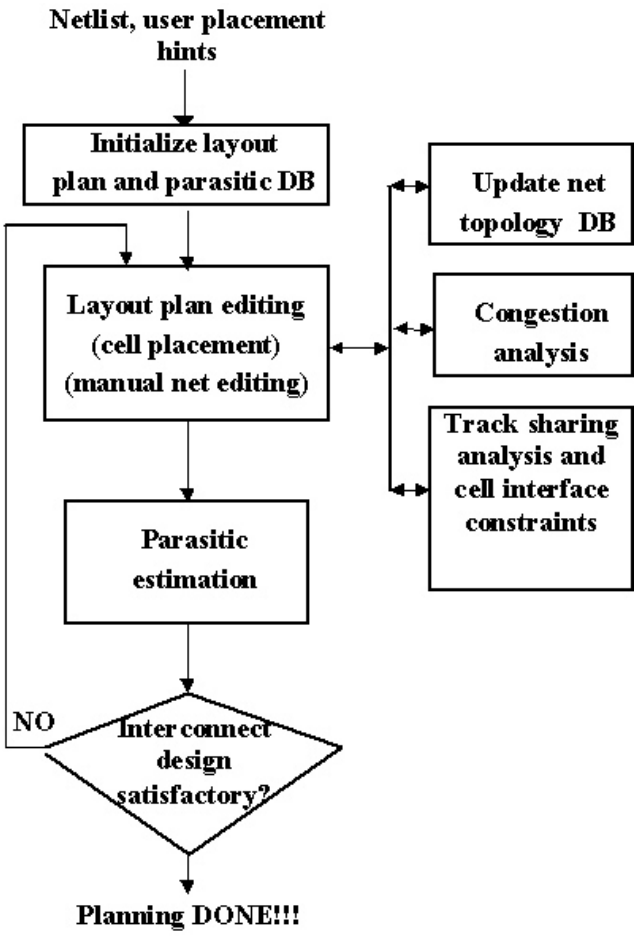

Figure 7: Layout planning design flow

4.3. Benefits of Layout Planning

From a recent Intel® microprocessor design project, effort analysis indicated that when early layout planning is carried out, the amount of re-design and rework required is reduced by approximately half. This benefit is expected to be even more significant in the future when we have more stringent and complex design requirements.

4.4. The Challenges in Datapath Auto Placement

Placement is a very critical stage in the datapath layout design flow as it can make or break regularity, area, and timing specifications. If a designer starts out with a bad placement, it is extremely difficult for the router to make up for it.

The main difference between datapath placement and Random Logic Synthesis (RLS) placement is the need to maintain regularity and hierarchy. Maintaining regularity in datapath placement offers several advantages. The circuit designer can rely on the regularity to use his intuition about critical paths. Regular layout tends to be more dense because of the reduction in the number of jogs/bends and because the designers can spend more time on optimizing one bit-slice. Regularity and hierarchy in layout are also very useful in reducing ECO time.

Timing and area constraints also tend to be much more critical for datapath blocks than RLS blocks. Unlike RLS blocks, datapath blocks are often made from custom designed cells that don't come with all the timing characterization data. This poses additional challenges for timing-driven placement algorithms.

Not all datapaths are fully regular, and they show differing amounts of irregularity, something the datapath auto-placement algorithm must contend with.

Traditional auto placement techniques, based on mathematical programming (usually with a quadratic objective function) or simulated annealing, can be modified to deal with the unique requirements of datapath placement with varying degrees of success. Techniques based on quadratic programming tend to be faster, but the rigid formulation makes it difficult to directly model the regularity requirements.

V. RC Estimation

Increased use of noise sensitive dynamic circuits, lower supply voltages, and increasing current density have made more extensive interconnect analyses a requirement in the design process. Such effects must be modeled at all stages of the design process. Estimation of the effects of interconnects and device parasitics must be accurate and consistent at all stages of the design in order to avoid unnecessary design iterations. Accurate parasitic estimation in the datapath design flow depends on both the prediction of the physical properties of the interconnects and devices (topology, routing layers, density, device layout, etc.) and on the accurate modeling of the parasitic effects of the devices and routing.

5.1. Layout Estimation Techniques

A wide range of layout estimation techniques are in use in design tools, ranging from wire length estimation to detailed net topology estimation. Such techniques are based on a set of rules, such as default routing layers, widths, and spacing, and on net topology generation algorithms such as a minimum spanning tree or Steiner tree (minimum length routing tree with horizontal and vertical wires). Some estimates may account for metal density or routing congestion constraints. The accuracy of layout estimation is dependent on the state of the design data. Estimated layout based on a globally routed floorplan may be very close to the final detailed routing, while schematic-

based estimates using little physical design data may correspond poorly with the final design. Thus, the quality of the estimated layout is highly dependent on how well the datapath design tools provide an early estimation of the physical design.

5.2. Parasitics Modeling Techniques

The modeling of process-related effects is a fairly mature field, with a wide range of tools, models, and techniques in use. Models range from empirical, easy to evaluate equations [20], to computationally intensive field solvers [19]. A wide range of parasitics' modeling tools are available both commercially and from universities. Commercially available tools provide reasonable accuracy (within 10% of field solvers) on large designs, and field solver accuracy is possible on a per-net basis [21]. Most commercial tools handle only post-layout parasitic extraction and are suitable only for final verification of designs. Many analysis tools and physical design tools (such as circuit analysis tools or global routers) have built-in parasitic estimation capability to estimate the effect of interconnect parasitics, but such tools cover only a part of the design process. The models used by these tools may not make use of all available design data, and inconsistency in the parasitics models used by different tools may result in poor convergence of the design and increased design cycle time. In addition, the built-in estimation may not accurately model cross capacitance and may not easily extend to new types of analysis required in the design flow.

5.3. Parasitic Estimation

Our work on parasitic estimation in the datapath design process focuses on accurate, consistent parasitic estimation at all stages of the design. The first and perhaps most important element in the accurate estimation of parasitics is the datapath design flow itself, particularly the close interaction and sharing of design data between the tools in the flow. The next element is the flexibility of the parasitic estimation tool to handle design data at all stages of completion, and the ability to support the wide range of constraints and assumptions required at each stage of the design. An extensive net specification system is an integral part of the design tool suite, providing designers the ability to specify a wide range of properties on the nets in the design. These net specifications are used by the parasitic estimation capability to ensure that the parasitic estimates accurately reflect the designer's intentions.

The parasitic estimation capability works by using all available design data to build a description of each of the nets in the design as well as the environment surrounding the nets. The estimator is based on a common representation of the layout and connectivity data. Design data from various tools in the datapath design flow are translated into this representation. Before the final stage of the design, when the layout is complete, the data for the nets will be incomplete. For example, in the floorplanning stages, the net's routing topology will not be available. Using a range of assumptions, the missing net data will be estimated. These assumptions may be tuned to match a particular layout design style. A key advance over existing parasitic estimation tools is that we are able to make use of any real layout data that exists. Estimated layout is used only when necessary to complete a net's representation. Since even drawn layout may not represent the final design, the estimator provides the capability to ignore any existing layout and replace it with estimated layout.

Next, the appropriate model is used to estimate the parasitics for each net. In our datapath design flow, the parasitic estimation tool is able to make use of a mix of input data sources and assumptions. We have developed a consistent set of models of varying accuracy that are built into the estimation tool. These models estimate interconnect and device resistance and capacitance, including cross capacitance. The estimator applies the appropriate model based on the source of the input data. The model used depends on the confidence of the original design data. Higher accuracy models are used when there is higher confidence in the design data. For example, an estimation based on a floorplan for a preliminary schematic need not use a high-accuracy model since the design is likely to change, while in the later stages of the design when much of the layout is complete, a high-accuracy model is needed to estimate cross capacitance between the nets.

The flexibility provided by the parasitic estimator allows the same tool to be used at all stages of the datapath design and helps ensure consistent results of the analyses at each stage of the design. It should be emphasized that the effectiveness of the parasitic estimator is dependent on the consistency of the results of each of the stages of the design process in the sense that the design at any stage provides an accurate estimation of the next stage and a reasonable early estimate of the final design. As shown in the other sections of this paper, this will be the case.

5.4. Results

Our initial results have shown that the parasitic estimator provides superior accuracy compared to the estimators used in existing point tools in the current datapath design flow. The benefits of the estimator will increase further when it is consistently used in the complete datapath design flow.

VI. Layout Cell Generation

The research in the field of cell synthesis was started more than 15 years ago [42]. Most of this research has focused on the generation of so-called 1-dimensional (1D) layouts when transistors are arranged in a linear fashion to minimize the number of diffusion breaks. First approximated algorithm for this layout style has been suggested by Uehara and VanCleemput [42]. Maziasz and Hayes [37] presented the first optimal algorithm.

Unfortunately 1D layout style is suitable only for small cells with fully complementary non-ratioed series-parallel CMOS circuits. Multiple attempts to extend this style have been made to handle more complicated circuit structures [23, 24, 31, 33, 36, 39].

Analysis of manually drawn layouts shows that "two-dimensional" (2D) layouts must be generated. Various approaches have been taken to address this problem [26, 27, 28, 29, 30, 40, 41, 43].

Though some of these leaf-cell layout systems have been applied successfully in ASIC flows, no commercially available system today has the capabilities to address the requirements of a custom design flow such as microprocessor design, where layout cell design involves a number of complex requirements. As chip designs approach GHz frequencies, reliability verification (RV) constraints, arising from the electro-migration and self-heat phenomenon, have also proven to be a critical factor in the generation of leaf-cell layouts.

6.1. Feature Requirements

A cell layout generation system is being looked into by us [44]. The system has to enable automated layout generation to produce cells that are optimized for various constraints such as density, performance, RV, and power. Its goal is to increase cell design productivity.

The system should include the following features:

- Ability to handle several hundred devices with various types of top-down constraints such as pre-routes, keep-out regions, pin/port preferred locations, etc.
- Easy configuration for various design domains (standard cell libraries, datapath bit-cells and bit-slice synthesis, custom cell design, etc.) and different circuit design methodology. Users should be able to define their own cell architecture rules.
- True 2D placement with RV constraints that allows simultaneous placement of cell instances and devices.
- Automatic stack and/or device-based legging with optional user control.
- Incremental area routing.
- Incremental compaction with different types of gridding constraints.
- Link with schematic editor.
- Powerful ECO mode / family generation / process migration capabilities.
- On-line RV estimation, DRC, and OpenChecker.
- Integrated with a layout editing system to allow manual intervention at any stage, ranging from push-button mode (fully automatic) to an interactive mode with unlimited manual intervention.

6.2. System Overview

In order to implement this layout generation system, five main components are required: a placer, a router, an RV analyzer, a compactor, and a family generator and change manager. A layout generation flow can be built around these five components (Figure 8). This flow can either be fully automated, or it can be guided and enhanced by a layout designer wherever required.

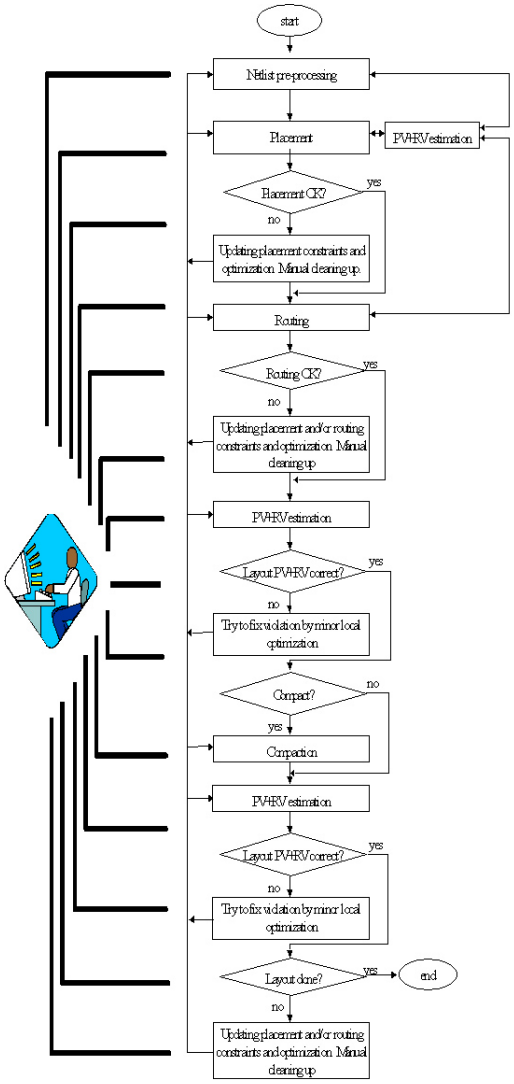

Figure 8: ***Layout synthesis flow***

6.2.1. Placer

In the generation of a layout for any schematic, a large amount of effort is spent in transferring the netlist to the layout editor, ordering the devices, and then determining the best placement for those devices. The placer has to handle a capacity for several hundred devices and be able to do two-dimensional device placement. It should also have provisions for top-down constraints, RV constraints, and an incremental placement capability.

6.2.2. Router

Once the devices have been placed, the connections between them have to be made. These are done by the routing module. Manual pre-routing of critical nets is allowed, and often encouraged, to meet strict timing or port location guidelines. The router interacts with the RV estimator to deduce the optimal routing shapes for critical nets based on given RV constraints. Once the optimal routing topologies have been determined, the actual routing itself is done by a detailed router. To improve routing quality, the router module refines the placement based on congestion analysis.

6.2.3. RV Analyzer

At different stages of the work flow, RV estimations are required to produce layouts that are optimized for reliability constraints. The RV estimator is based on worst-case current analysis through static modeling of current switching. It has a built-in current-solving engine that traverses through nets to compute worst-case interconnect currents from the switching of the device stacks. Based on the results of the analysis, the module identifies objects that are electro-migration and self-heat limited. The RV analysis can potentially lead to a re-ordering of devices, a change in routing topology, or a change in wire and via geometries.

6.2.4. Compactor

This is used to compact the area and resolve design rule violations, as well as for putting pins on grid for supporting the routing flow at the next level of design hierarchy. It can be configured by a wide range of options to support a specific working flow.

6.2.5. Family Generator and Change Manager

While generating cell libraries, several cells of similar topologies need to be created, the differences usually being ones of device sizing, with minor changes in schematic, legging, etc. The same situation is also encountered if the schematics are revised after the layout has been done. Since such changes don't modify the fundamental layout topology, we can generate subsequent cells from a starting prototype or template. This is done by creating a mapping between the netlists of the template and the desired cell, followed by resizing, and adding or deleting devices or legs as necessary. Using the family generation module, the layout designer only has to lay out a couple of representative cells of a cell family. The layouts for all the other members of the same cell family are then generated automatically. This feature can also be used for process migration.

Each of the above steps are independent of each other: for example, the devices may be manually placed and then automatically routed, or, cells drawn manually may be used as templates for family generation, and so on. This ensures improved layout design productivity without compromising the layout quality.

6.3. Results

Initial usage of a prototype version of the cell generation system at Intel Corporation shows significant productivity improvement over manual design for various kinds of cells, while meeting all layout quality requirements such as density, reliability, power, and timing.

VII. Future Challenges

It is expected that with continuous process technology advancement and the growing need for higher performance chips, the problems in datapath design will continue to increase and become more complex. Granted, not all problems are known or understood at this time. There are a number of problems that we are dealing with currently, which will get much worse in the future. They are as follows:

- *Handling of coupling noise problems (primarily due to capacitive coupling).* A substantial amount of effort is currently required to verify and correct the design to ensure correct silicon behavior. Techniques to generate correct-by-construction noise problem-free circuit and layout are essential.
- *Timing analysis to include effects of noise (power noise, capacitive coupling, and inductive coupling)*. Guard banding (in timing analysis cycle time or interconnect capacitance) is often used to account for the effect of noise. However over-conservatism would result if the guard banding is done for the worst-case scenario. If some statistical averages are used in guard banding, there might be serious escapes, which can cause problems in silicon. Hence, it is necessary to have the ability to include the effects of noise accurately in timing analysis.
- *New circuit techniques*. Traditional static CMOS and domino logic circuits have worked well so far. However, with the continuous decrease in power supply voltage and the increased demand in chip performance, new circuit design styles have to be investigated to achieve a better delay-power product and to meet other design requirements.

VIII. Conclusion

In this paper, we have presented the challenges in datapath design and our ideas to meet these challenges through datapath logic synthesis, layout planning, interconnect RC estimation, and layout cell generation. We believe that datapath design requires substantially more automation to be able to meet future requirements: "system on a chip" and demand for higher performance and deep sub-micron geometries. We hope that this paper stimulates more interest in both academic and commercial CAD arenas to tackle the problems in high-performance datapath design.

Acknowledgments

We thank Nagbhushan Veerapaneni who contributed to the paper in the area of auto placement. We also thank Marian Lacey, Mysore Sriram, Bharat Bhushan, and Lin Chao who reviewed this paper and gave valuable feedback.

References

[1] D.E. Hoffman, "Deep Submicron Design Techniques for the 500MHz IBM S/390 G5 Custom Microprocessor," Proc. of ICCD 1998.

[2] S. Posluszny, "Design Methdology for a 1.0 GHz Microprocessor," Proc. of ICCD 1998.

[3] S. R. Arikati and R. Varadarajan, "A signature based approach to regularity extraction," *Proc. of ICCAD*, November 1997, pp. 542-545.

[4] M. Hirsch and D. Siewiorek, "Automatically extracting structures from a logical design," *Proc. of ICCAD*, November 1988, pp. 456-459.

[5] G. de Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw Hill, New York, 1990.

[6] E. Detjens, *et al*., "Technology mapping in MIS," *Proc. of ICCAD*, November, 1987, pp. 116-119.

[7] K. Keutzer, Dagon, "Technology binding and local optimization by DAG matching," *Proc. of DAC*, June 1987.

[8] M. R.Corazao, *et al*., "Performance optimization using template matching for datapath-intensive high-level synthesis," *IEEE Trans. on CAD*, 15(8), August 1996, pp. 877-887.

[9] G. Odawara, *et al*., "Partitioning and placement technique for CMOS gate arrays," *IEEE Trans. on CAD*, May 1987, pp. 355-363.

[10] R.X.T. Nijseen, and C. A. J. van Eijk, "Regular layout generation of logically optimized datapaths," *Proc. of ISPD*, 1997, pp. 42-47.

[11] J. M. Rabaey, *et al,*. "Fast prototyping of datapath-intensive architectures," *IEEE Design and Test of Computers*, June 1991, pp. 40-51.

[12] D. S. Rao and F. J. Kurdahi, "On clustering for maximal regularity extraction," *IEEE Trans. on CAD*, 12(8), August 1993, pp. 1198-1208.

[13] A. Chowdhary, *et al*., "A general approach for regularity extraction in datapath circuits," *Proc. of ICCAD*, November 1998, pp. 332-339.

[14] A. Chowdhary, *et al*., "Extraction of functional regularity in datapath circuits," *IEEE Trans. on CAD*, submitted November 1998.

[15] A. Chowdhary, *et al*., "A systematic approach for regularity extraction," *U.S. Patent*, filed November 7, 1998.

[16] B. Krishna, C.Y.R. Chen, and N. Sehgal, "Technique for Planning of Terminal locations of Leaf Cells in Cell-Based Design," *Proc. 11th International Conference On VLSI Design*, pp. 53-58, 1998.

[17] Amnon Baron Cohen, Michael Shechory, "Track Assignment in the Pathway Datapath Layout Assembler," *Digest of Technical Papers 1991 IEEE International Conference on Computer-Aided Design*, pp. 102-105, 1991.

[18] J. Cohn, L. Pillage, and I. Young, "Tutorial 4: Digital Circuit Interconnect: Issues, Models, Analysis, and Design," *IEEE/ACM International Conference on CAD-94.*

[19] J. R. Phillips and J. White, "A Precorrected-FFT Method for Capacitance Extraction of Complicated 3-D Structures," *Proc. ICCAD-94*, pp. 268-271.

[20] N. Delorme, M. Belleville, and J. Chilo, "Inductance and Capacitance Analysis Formulas for VLSI Interconnects," *Electronics Letters*, vol. 32, no. 11, May 1996.

[21] Y.L. Le Coz , R.B. Iverson, H.J. Greub, P.M. Campbell, and J.F. McDonald, "Application of a Floating-Random-Walk Algorithm for Extracting Capacitances in a Realistic HBT Fast-Risc RAM Cell," *Proc. 11th International VLSI Multilevel Interconnection Conference*, Santa Clara, CA, pp. 342-4, June 1994.

[22] D.G. Baltus and J.Allen, "SOLO: A generator of efficient layouts from optimized MOS circuit schematics," *Proc. 25th ACM/IEEE Design Automation Conferenec,* pp. 445-452, June 1988.

[23] B.Basaran, "Optimal Diffusion Sharing in Digital and Analog CMOS Layout," Ph.D. Dissertation, Carnegie Mellon University, CMU Report No. CMUCAD-97-21, May 1997.

[24] J.Burns and J.Feldman, "C5M: A Control Logic Layout Synthesis System for High-Performance Microprocessors," *Proc. ISPD'97*, pp. 110-115.

[25] C.C.Chen and S.L.Chow, "The layout synthesizer: An automatic netlist-to-layout system," *Proc. 26th ACM/IEEE Design Automation Conference,* pp. 232-238, June 1989.

[26] S.Chow, H.Chang, J.Lam, and Y.Liao, "The Layout Synthesizer: An Automatic Block Generation System," *Proc. CICC* 1992, pp. 11.1.1-11.1.4.

[27] J.Cohn, D.Garrod, R.Rutenba, and L.R.Carley, *Analog Device-Level Layout Automation*, Kluwer Academic Publishers, Boston MA, 1994.

[28] M.Fukui, N.Shinomiya, and T.Akino, "A New Layout Synthesis for Leaf Cell Design," *Proc. 1995 ASP-DAC*, pp. 259-263.

[29] A.Gupta and J.Hayes, "Width Minimization of Two-Dimensional CMOS Cells Using Integer Linear Programming," *Proc. ICCAD* 1996, pp.660-667.

[30] A.Gupta and J.Hayes, "CLIP: An Optimizing Layout Generator for Two-Dimensional CMOS Cells," *Proc. 34th DAC* 1997, pp.452-4557.

[31] A. Gupta and J. Hayes, "Optimal 2-D Cell Layout with Integrated Transistor Folding," *Proc. ICCAD* 1998, pp.128-135.

[32] M. Guruswamy, R. Maziasz, D. Dulitz, S. Raman, V. Chiluvuri, A. Fernandez, and L. Jones "CELLERITY: A Fully Automatic Layout Synthesis System for Standard Cell Libraries," *Proc DAC'97*, pp. 327-332.

[33] Y-C. Hsieh, C-Y. Hwang, Y-L. Lin, and Y-C. Hsu, "LiB: A CMOS cell compiler," *IEEE Transactions on Computer Aided Design,* Vol. 10, pp. 994-1005, August 1991.

[34] Chi Yi Hwang, Yung-Ching Hsieh, Youn-Long Lin, and Yu-Chin Hsu, "An Efficient Layout Style for Two-Metal CMOS Leaf Cells and its Automatic Synthesis," *IEEE Transactions on Computer Aided Design*, Vol. 12, pp. 410-423, March 1993.

[35] M. Lefebvre, C. Chan, and G. Martin, "Transistor placement and interconnect algorithms for leaf cell synthesis," *EDAC-90*, pp. 119-123, 1990.

[36] M. Lefebvre and D. Scoll, "PicasoII: A CMOS Leaf Cell Synthesis System," *Proc. 1992 MCNC Intl. Workshop on Layout Synthesis*, Vol. 2, pp. 207-219.

[37] R.Maziasz and J.Hayes, "*Layout Minimization of CMOS Cells*," Kluwer Academic Publishers, Boston, 1992.

[38] C.L. Ong, J.T. Li and C.Y. Lo, "GENAC: An automatic cell synthesis tool," *Proc. 26th ACM/*

IEEE Design Automation Conference, pp. 239-244, June 1989.

[39] C.Poirier, "Excellerator: Custom CMOS Leaf Cell Layout Generation, " *IEEE Trans. on CAD, 8(7)*, July 1989, pp. 744-755.

[40] S. Saika, M. Fukui, N. Shinomiya, and T. Akino, "A Two-Dimensional Transistor Placement Algorithm for Cell Synthesis and its Application to Standard Cells," *IEICE Trans. Fund., E80-A(10)*, Oct.1997, pp. 1883-1891.

[41] K. Tani, K. Izumi, M. Kashimura, T. Matsuda, and T. Fujii, "Two-Dimensional Layout Synthesis for Large-Scale CMOS Circuits," *Proc ICCAD 1992*, pp.490-493.

[42] T. Uehara and W.M. VanCleemput, "Optimal Layout of CMOS Functional Arrays*," IEEE Transactions on Computers, C-30(5),* May 1981, pp. 305-312.

[43] H. Xia, M. Lefebvre, and D. Vinke, "Optimization-Based Placement Algorithm for BiCMOS Leaf Cell Generation," *IEEE J.Solid State Circ., 29(10)*, October 1994, pp. 1227-1237.

[44] B.Basaran, K.Ganesh, A.Levin, R.Lau, M.McCoo, S.Rangarajan, and N.Sehgal, "GeneSys - A Layout Synthesis System for GHz VLSI Designs," Proc. 12th International Conference on VLSI Design, January 1999, pp. 458-452.

Authors' Biographies

Tim Chan is currently a CAD tool system architect at Intel. He received his B.Sc. and M. Phil. in electrical engineering from the University of Hong Kong in 1980 and 1984 respectively. He joined Intel in 1990 as a senior VLSI designer. His technical interests include high-speed circuit design, microprocessor design methodology, and logic synthesis. His e-mail is tim.w.chan@intel.com.

Amit Chowdhary is a senior CAD engineer at Intel. He received his Ph.D. in computer science and engineering from the University of Michigan, Ann Arbor in 1997. He joined Intel in 1997 and is working on the datapath automation project in Design Technology (Microprocessors Products Group). His technical interests include high-level and logic synthesis, technology mapping, and timing analysis. His e-mail is amit.chowdhary@intel.com.

Bharat Krishna is a senior CAD engineer in the Microprocessors Product Group at Intel. He received an M.S. in computer engineering from Syracuse University in 1994 and a B.Sc. in electrical engineering from the University of Khartoum, Sudan in 1991. He has been working in Intel since 1995, and he is the project leader for the layout planner tool. His interests include datapath layout automation and VLSI routing. His e-mail is bharat.krishna@intel.com.

Artour Levin is a staff CAD engineer at Intel. He received his MS in Math from Belarus State University in 1986 and Ph.D. in Computer Science from the same University in 1990. He joined Intel in 1995 and is working on CAD tool and methodology development in the Microprocessors Product Group. His interests include discrete mathematics, combinatorial optimization, CAD algorithms and design methodology. His e-mail address is alevin@scdt.intel.com.

Gary Meeker Jr. is a senior CAD engineer at Intel. He received his BSEE from Carnegie Mellon University in 1990 and his MSEE from UC Berkeley in 1994. He joined Intel in 1994 and is working on CAD tool and methodology development in the Microprocessors Product Group. His interests include parasitic extraction and estimation tools, algorithms, and models. His e-mail is gmeeker@scdt.intel.com.

Naresh Sehgal is currently managing the Datapath Tool Development Group for next-generation processor design at Intel. He received his B.S. in EE from Punjab Engineering College in India, followed by an M.S. and Ph.D. in computer engineering from Syracuse University, NY. Naresh has been with Intel since 1988, and his research interests include CAD algorithms and design methodology. His e-mail is naresh.sehgal@intel.com.

CAD Design Flows Development in a Cross-Platform Computing Environment

Shesha Krishnapura, Computing Technology/Design Technology, Intel Corp.
Ty Tang, Computing Technology/Design Technology, Intel Corp.
Vipul Lal, Computing Technology/Design Technology, Intel Corp.

Index words: NT* , UNIX*, mixed-flow, cross-platform

Abstract

With the advent of low-price, high-performance Intel® architecture workstations together with Microsoft* Windows NT* operating systems (referred to as IA-NT from here on) that support Microsoft productivity tools, the IA-NT workstation has become the preferred desktop for CAD design engineers. However, due to the complexity of migrating UNIX*-centric legacy CAD tools and scripts to an NT environment, a mixed operating system platform for CAD design has become a computing reality. This paper describes the innovative technical solutions for a production-capable NT-UNIX cross-platform CAD design flow environment for development, maintenance, and deployment activities. Although the target systems chosen are the ones used in Design Technology at Intel, our solutions are applicable to other cross-operating systems.

The NT-UNIX platform poses various technical challenges when developing the CAD design flows consisting of tools from both platforms. These tools have to work together on a shared design database while effectively utilizing common infrastructure scripts, despite the fact that each computing platform supports a different scripting environment.

To meet some of these challenges, we developed two technologies that allow seamless integration of software, designed for either the UNIX or NT platform, into a platform-independent production usage environment. These two technologies have been used to port more than 45 tools made up of more than 3,000,000 lines of code from UNIX to NT, and to execute more than 1,000 test flows, as well as to develop a few mixed NT-UNIX applications.

Introduction

Traditionally, Intel has been using high-end UNIX*-based RISC workstations for microprocessor design activity. However there are emerging compelling reasons why this traditional design environment should change to incorporate the IA-NT workstation. The main reasons for this change are as follows:

1. The advent of low-priced, high-performance Intel architecture workstations coupled with Windows NT* operating systems (IA-NT), which make IA-NT a formidable alternative to UNIX-RISC workstations.
2. The maturity of NT towards a stable, scalable operating system that supports high-end CAD applications.
3. Next-generation Intel® CAD tools are moving from the legacy single CAD design environment, which is driven by scripts through command line interface, to a new kind of CAD environment. This new environment incorporates CAD design tools and office productivity tools into a tightly integrated visual cockpit that uses modern distributed computing components and Internet-driven technology that support multiple simultaneous CAD design environments. The IA-NT workstation provides an excellent development and design environment for these new-generation CAD applications.

In reality, we cannot convert the existing UNIX-RISC-based design flow to an IA-NT base in a single step due to the following reasons:

1. Many of the design automation tools are based on UNIX-centric scripts that are not easily ported to an NT environment.

2. Some of the internal CAD tools are tightly integrated with external CAD tools that are not available on NT.
3. The current design team skill set is UNIX-centric and would need to be updated for NT-centric design work.
4. Microprocessor design teams in the midst of projects cannot handle a change in environment due to the nature and complexity of such a change. This means that IA-NT can only be used on new projects.

The solution to converting to IA-NT therefore is to have a transition phase to support a production-capable mixed NT-UNIX design flow environment. To achieve this transition phase, the following needs to be done:

1. Build a robust NT-UNIX mixed computing environment with a shared file system. This will support the IA-NT Desktop with backend IA-NT and UNIX compute servers (see Figure 1).
2. Migrate high compute usage CAD design flows, comprised of tools and scripts from UNIX, to native NT and keep UNIX-centric legacy tools, which use low computing power, on UNIX.
3. Develop cross-platform utilities for production use for a mixed NT-UNIX design environment where CAD tools on NT and UNIX are used in a seamless fashion.
4. Develop next-generation CAD tools native to IA-NT.

In this paper, we limit our discussion to the NT-UNIX cross-platform environment. We outline the various challenges faced and the techniques employed for the production use of a mixed NT-UNIX environment for CAD tool development in Design Technology.

Overview of Existing Design Environment

The existing design environment at Intel is UNIX-centric. It consists of tightly integrated CAD tools, scripts, and design data that are in the order of tens of millions of lines of code. A significant portion of the tools and scripts are legacy codes that have been shared among generations of engineers and are hard to replace.

From a high-level point of view, the design tools, scripts, and data can be grouped into following four main categories, in which the first two categories are part of the CAD design tools development environment and the last two categories are part of the microprocessor design project environment:

1. *Design Tools:* A set of internally developed and external vendor CAD tools that are tightly coupled by UNIX-centric "glue" scripts into a tool suite.
2. *Gluing Utilities:* These are scripts and small programs that integrate the various tool components in a tool suite into a functional design environment for microprocessor designs at Intel. Some of the tasks include internal to external tools data format translation, data extraction from netlists, simulation, wave form analysis, and design database management.
3. *Microprocessor Design Project Utilities:* Sets of scripts and programs developed by design automation engineers in design projects to validate design logic, process design data, generate design models, analyze performance, etc.
4. *Designer Private Utilities:* Scripts and programs developed by individual design engineers to aid them in their work such as analyzing and filtering design data, generating test stimuli, running tools in a particular sequence, etc.

This complex environment is represented in Figure 2.

In the next sections we describe the technical challenges we faced while developing CAD design flows in a mixed NT-UNIX environment, and we outline some of the innovative technical solutions we adopted to overcome these challenges.

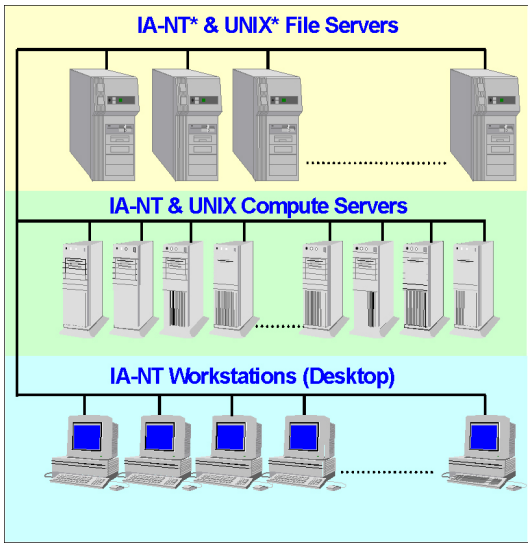

Figure 1: Simplified view of NT-UNIX mixed environment

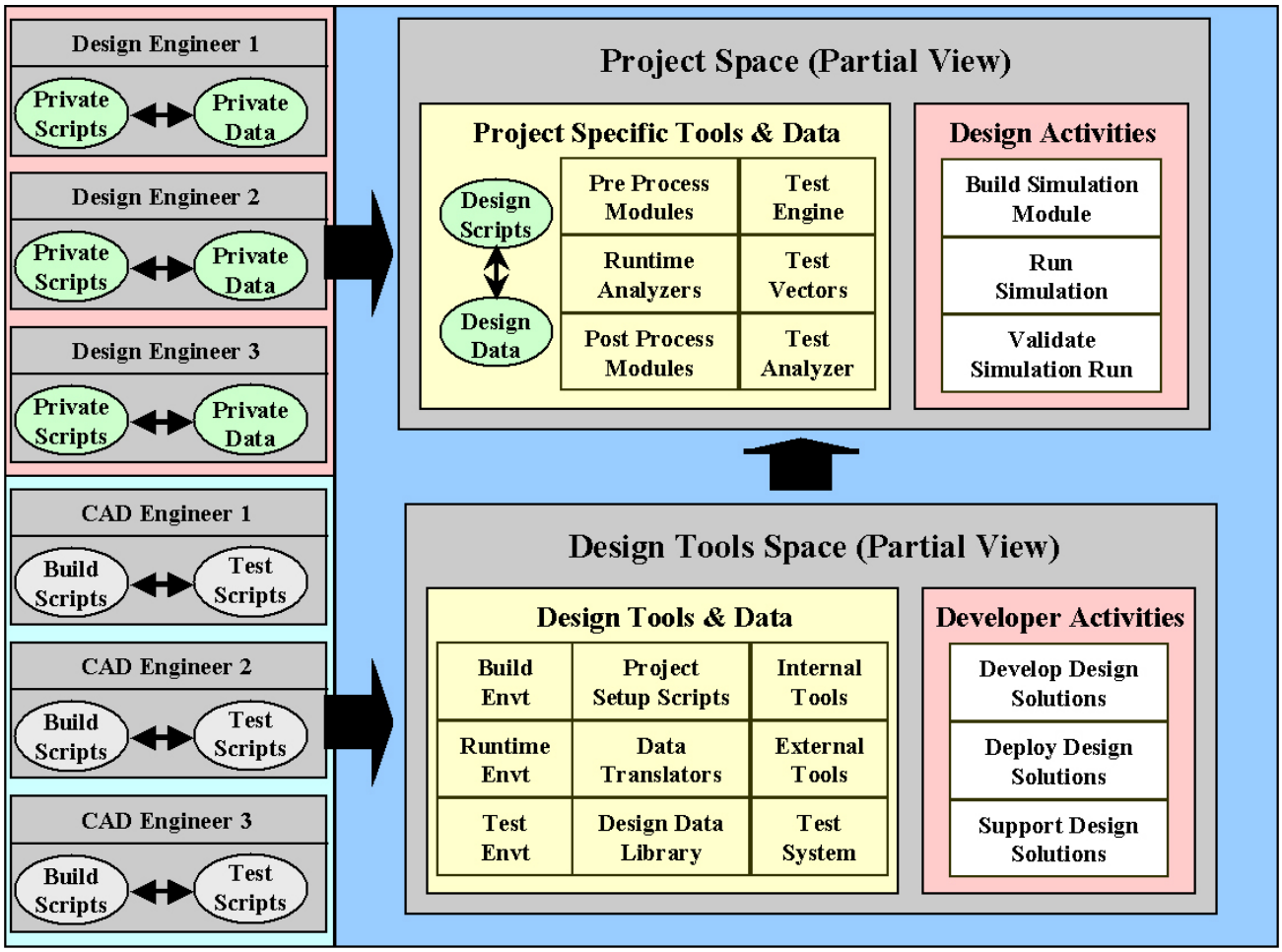

Figure 2: Simplified view of UNIX-based design environment

UNIX* to NT* Migration and NT-UNIX Cross-Platform Challenges

The complexity of the existing UNIX-based design environment at Intel makes it impractical to convert to a homogeneous NT environment in one step. The viable solution is to develop a heterogeneous UNIX-NT integrated design environment and convert more and more UNIX-centric components to IA-NT over time.

As with the migration of any operating system, the migration of an existing UNIX-based CAD design environment to a mixed NT-UNIX design environment presents us with a number of technical challenges of which the major ones are as follows:

1. New applications developed for or ported to NT depend on reusable components available only on UNIX. These reusable components include
   - external vendor tools and libraries
   - internal libraries that have not been ported or are not portable to NT
   - legacy design data in a database that can only be accessed on a specific UNIX platform
2. Design flows that execute a set of tools, all of which may not be available on a single platform.
3. The demand for common infrastructure scripts to drive the tools on both UNIX and NT is difficult to meet since the scripting environment on NT is not fully mature and is not 100% compatible with the UNIX scripting environment.
4. How to maintain a single test system and test vectors for cross-platform validation.

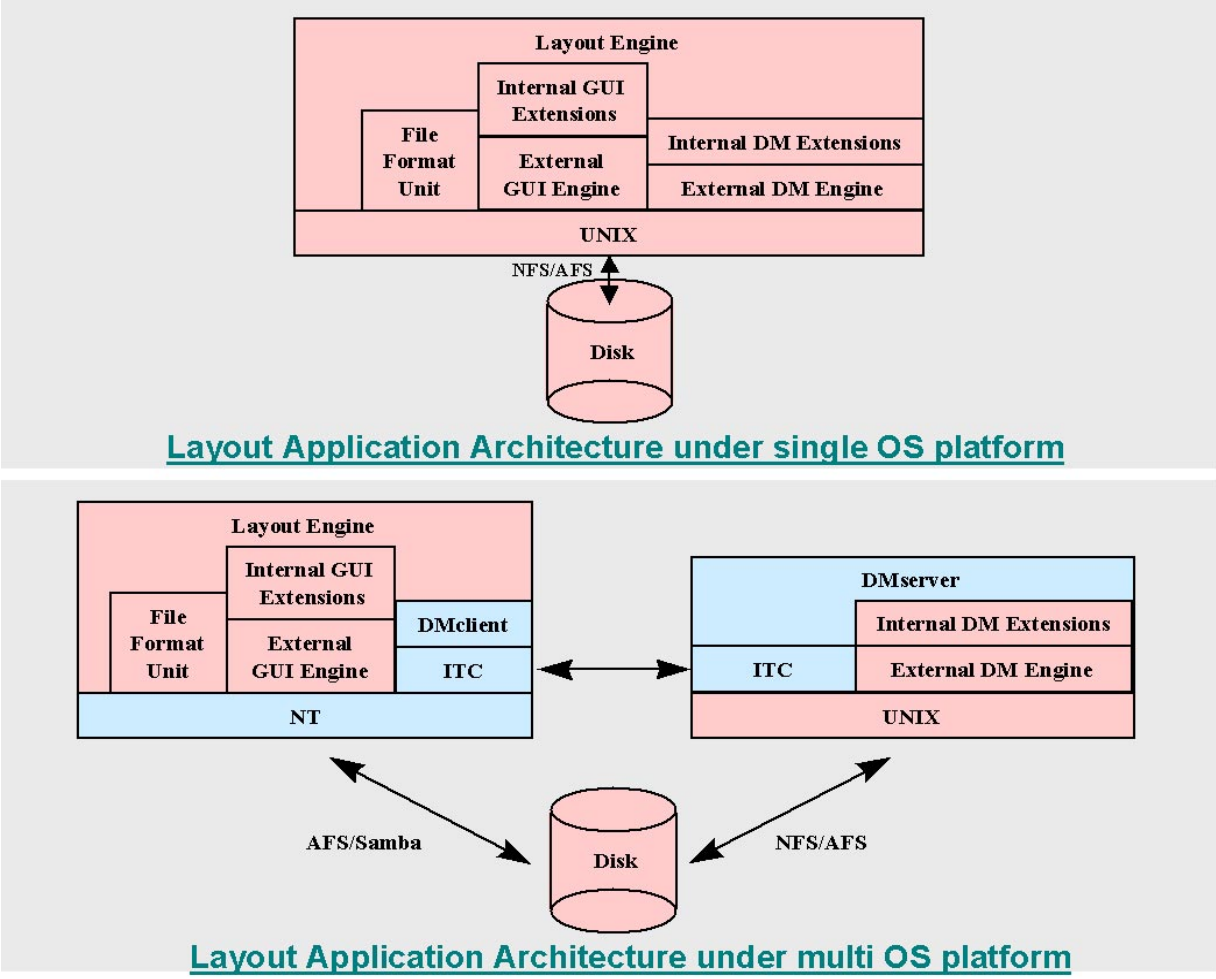

Figure 3: CAD layout application in single and multi OS platforms

Innovative Solutions for an NT-UNIX Cross-Platform Development Environment

The compile-time client-server model and the run-time client-server model were developed and integrated into the mixed NT-UNIX design environment to overcome the challenges detailed above. These models allow for the seamless integration of software developed for either UNIX or NT platforms into a platform-independent NT-UNIX CAD development and design environment.

In the following sections, we describe these technologies and outline the problems they solve, the architectural details, and our pilot results.

Compile-Time Client-Server Model

The compile-time client-server model allows UNIX*-based layered CAD applications to be migrated to IA-NT architecture in a situation where a complete migration to NT* is not practically feasible due to either the non-availability of vendor-provided components or non- portable internal legacy tools/libraries.

The UNIX-based layered application for a CAD layout capability is described in Figure 3. In this application, the Data Management (DM) engine is a vendor library integrated with an internal application not native to an NT platform. The figure shows the client-server application designed for an external DM engine using the innovative solution Inter Tool Communication Library (ITC) and integrated with a layout application. The architecture and functional components of the ITC library are described below.

The Inter Tool Communication Library for Compile-Time Client-Server Architecture

An important component of the compile-time client-server model is the ITC library, which provides inter-tool communication functionality for client-server applications.

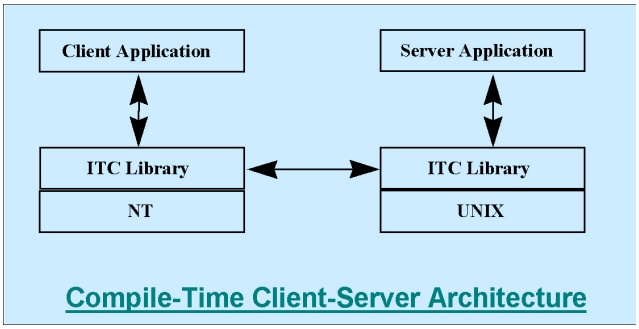

Figure 4: Compile-time client-server architecture

The aim of the ITC library is to provide the following:

- a reliable, recoverable, trackable, and efficient NT-UNIX communication channel with simple APIs for point-to-point inter-tool communication between a client running on an IA-NT system and a server running on a UNIX system
- transparent handling of Byte ordering between RISC and IA architecture
- transparent handling of NT-UNIX path conversion
- interface code to engineer compile-time client-server model for rapid implementation with built-in message build/extract capability and remote function execution mechanism

In this paper we do not focus on the implementation details of the ITC library APIs. Nonetheless, the next two figures clearly illustrate the interaction between an NT-client application and a UNIX server to remotely execute the extended features provided by libraries available only on UNIX. Figure 5 shows the interaction between the various ITC APIs on the NT client and the UNIX server describing the execution flow. Figure 6 shows a sample DMclient-DMserver application of which the UNIX-DMserver provides the NT-DMclient with complementary database access and management capabilities supplied by a UNIX-centric DM library not available on the IA-NT. The master server is capable of supporting multiple clients as shown in the figure.

The summary of the interaction between the client and server is as follows:

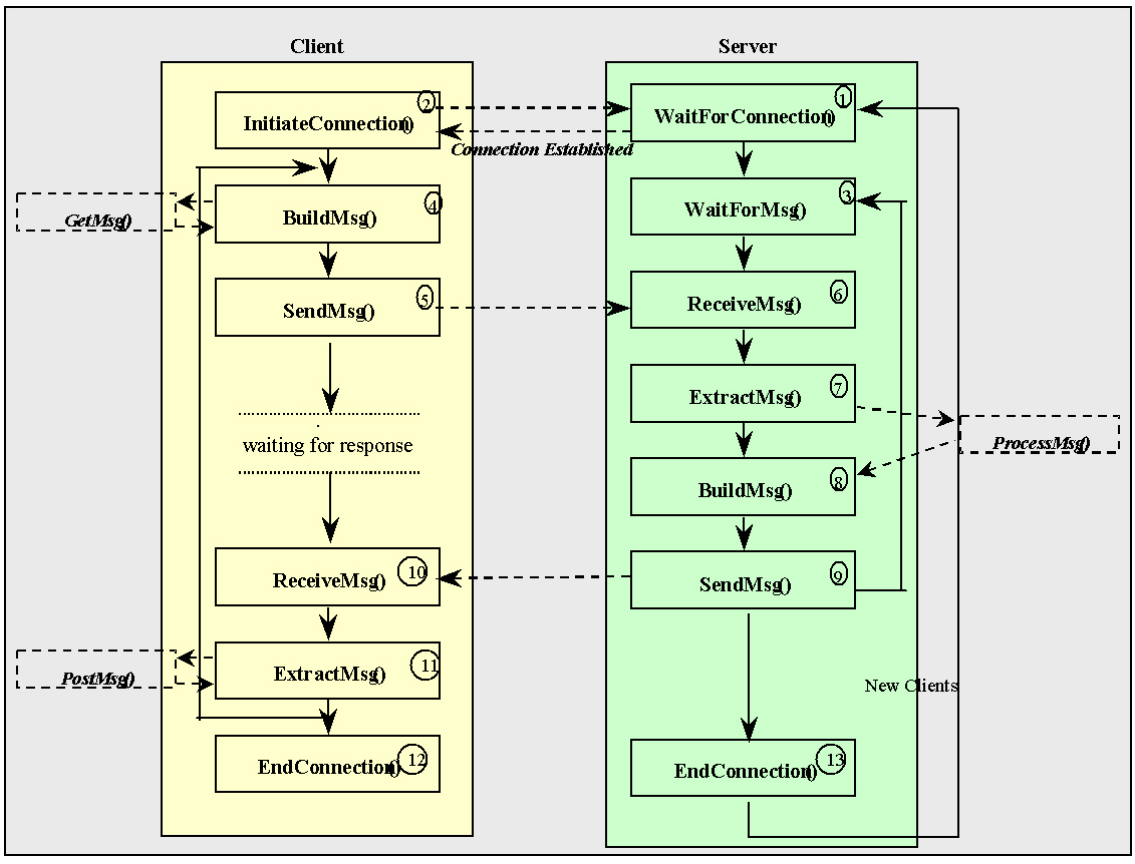

Figure 5: Interaction between NT client and UNIX server execution flow

- The client connects to the DMserver.
- The server forks off a slave server to process the client.
- The master server is free to handle incoming clients.
- The slave server, which is dedicated for each client, exits after servicing the client.

Over the past two years, the CAD organization at Intel has successfully migrated many tools to IA-NT and developed new IA-NT resident tools using this client-server technology.

The compile-time client-server technique is valid for the migration of any one platform to another platform. Until the complete set of native CAD tools and libraries are available to IA-NT, including the external vendor tools, this technology remains a good intermediate solution for the unavailable software components during UNIX to IA-NT migrations.

Run-Time Client-Server Model

Run-time client-server platform independent execution capability is a transition technology to enable the integration of a mixed NT*-UNIX* design environment. Its main aim is to address the very problems that prevent the design organization from moving directly to a pure IA-NT compute environment: not having the complete set of design CAD tools on IA-NT and having legacy UNIX-centric design data, infrastructure, and gluing scripts that cannot be ported to IA-NT.

The idea behind run-time client-server platform independent execution technology is to extend the remote procedure call concept to a remote execution environment. The result is a tool environment that allows transparent execution of CAD tools in a mixed NT-UNIX environment. In this cross-platform environment, the user will be working on an NT desktop and executing design commands from a remote UNIX Xterm using the same familiar design flow written in scripts. The user view is illustrated in Figure 7.

Run-Time Client-Server Architecture

The main components of the run-time client-server platform independent tool execution environment as shown in Figure 8 are a UNIX server daemon, an NT server service, and CAD tools and scripts that are set up to run in this environment.

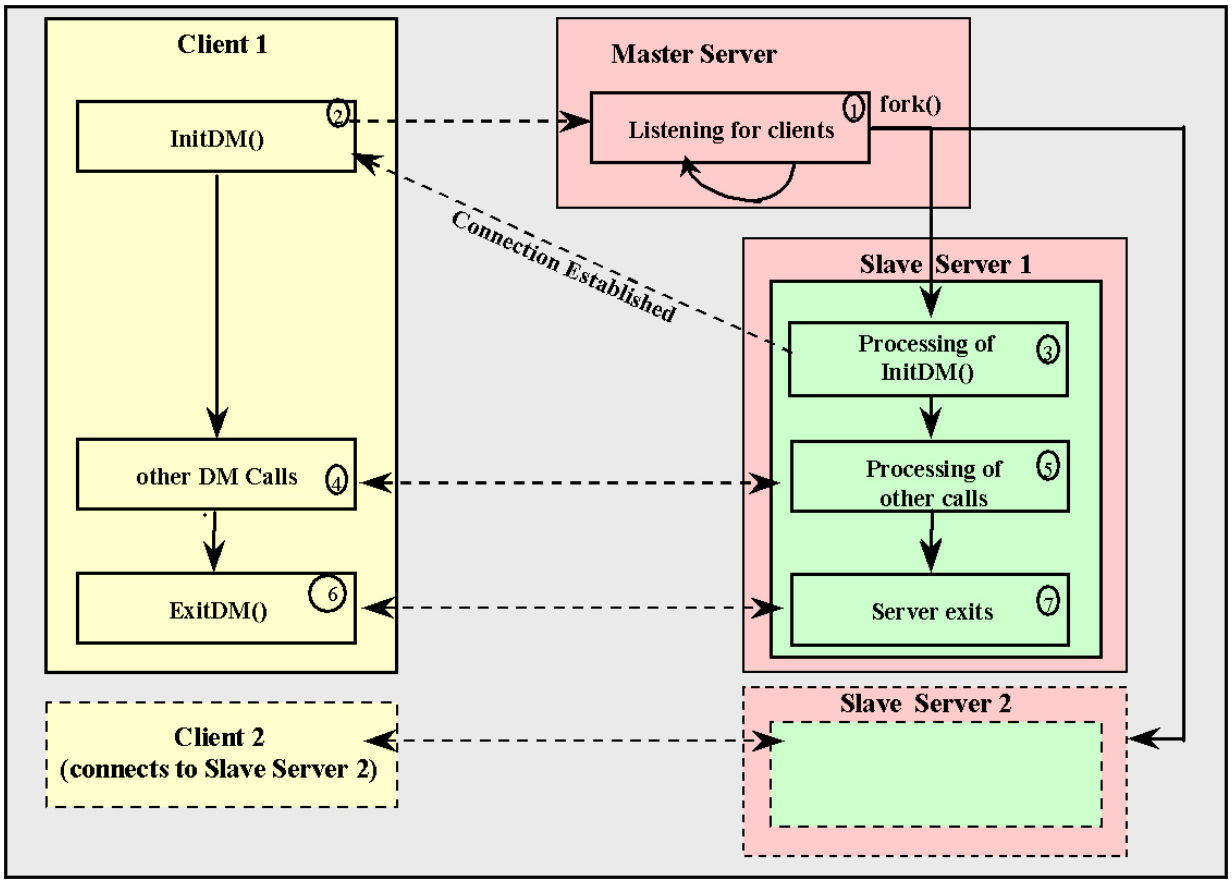

Figure 6: Execution flows between IA-NT DMclient application and UNIX DMserver

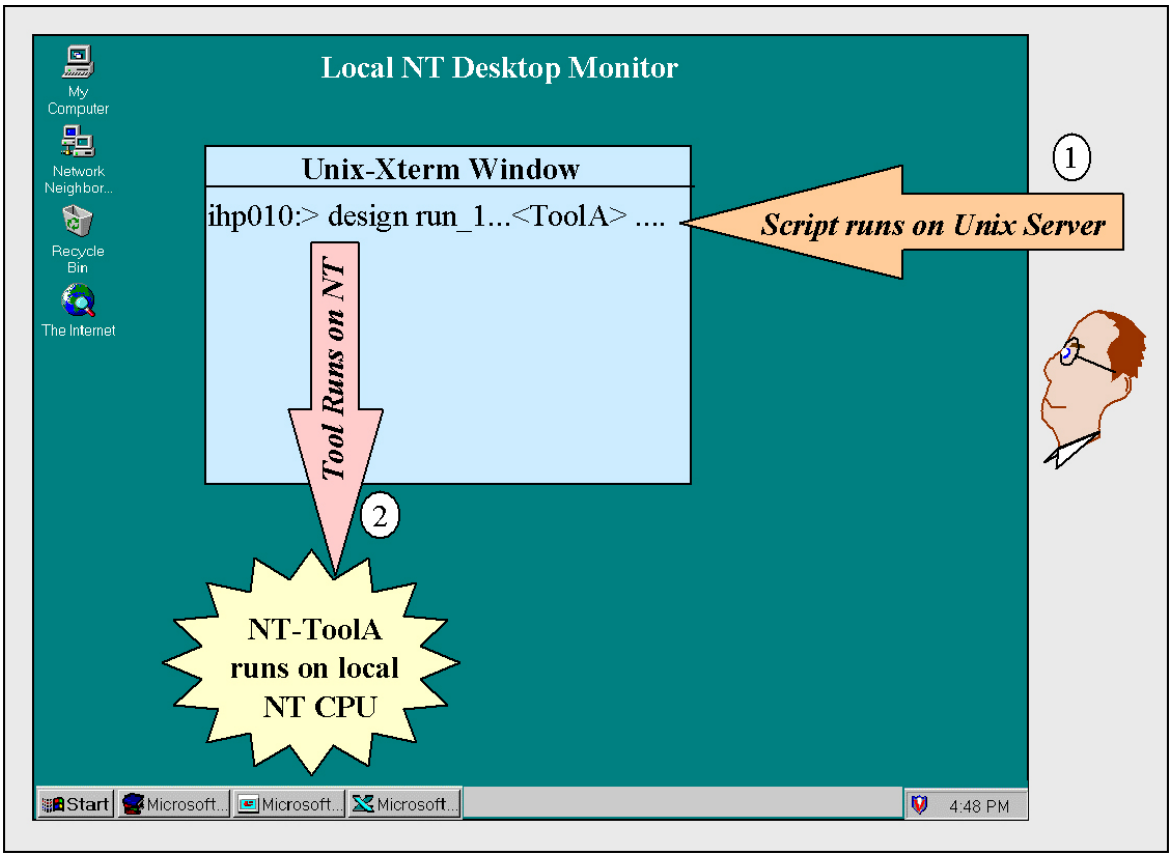

Figure 7: User view of run-time client-server platform independent execution environment

Every tool executable in this environment has two complementary components, namely the tool executable itself and the tool stub, each of which is running on different platforms. The tool stub is a wrapper that launches the tool executable on the remote system.

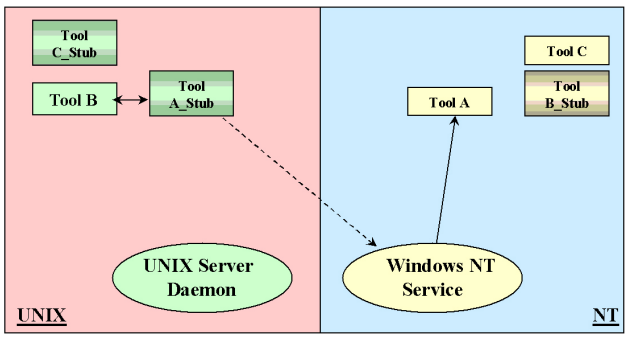

Figure 8: Overview of the run-time client-server platform independent execution environment

When a tool stub is executed on the local system, it takes a snapshot of the run-time environment on the local system and communicates this information to the server daemon (or service). The server impersonates the user, duplicates the run-time environment (with the necessary system path conversion and platform specific adjustment), and invokes the actual tool executable. The server also handles the routing of STDIO streams to the tool stub and passes the exit code of the tool executable to the tool stub. The tool stub exits with the same exit code.

A sample CAD design flow consisting of three tools working in a cross-platform execution environment using the above stated run-time client-server architecture is shown in Figure 9.

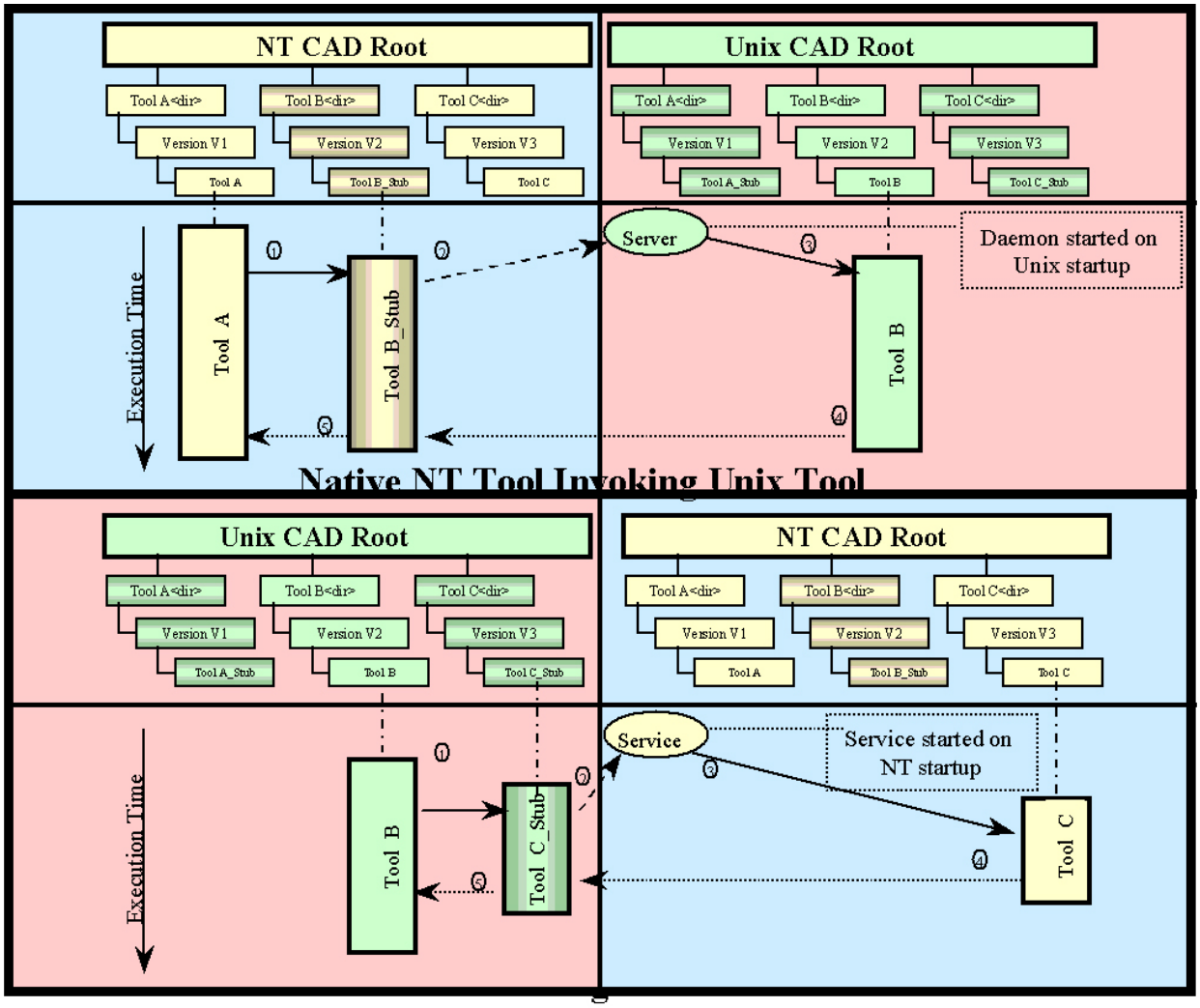

Figure 9: Flow execution using platform independent execution capability

The run-time client-server platform independent execution environment provides the following features to enable transparent remote tool execution:

1. Conversion and customization of the run-time tool environment. The run-time environment on the local system is duplicated on the remote system for tool execution.
2. NT and UNIX shared file-system support so that it can handle the tool stubs, executables, data, and output files residing on shared file systems.
3. Handles the redirection of standard input, standard output, and standard error streams between the tool stub on the local system and the tool executable on the remote system.
4. Provides the capability to execute binary files on the remote system (UNIX or NT).
5. Provides the capability to launch the scripts on the remote system (UNIX or NT).
6. Duplicates the shared file system credentials for transparent user access from local to remote system.
7. Impersonates the local system user on the remote system and executes the tool with the same user credentials.

Figure 10 details the architecture of the run-time client-server capability.

Over the past year, the CAD organization at Intel has successfully used this technology to validate more than 40 IA-NT ported CAD tools using the same test system from UNIX by running more than 1,000+ test flows in a seamless mixed NT-UNIX platform. These test flows also used some of the UNIX-centric tools that are not yet available on NT.

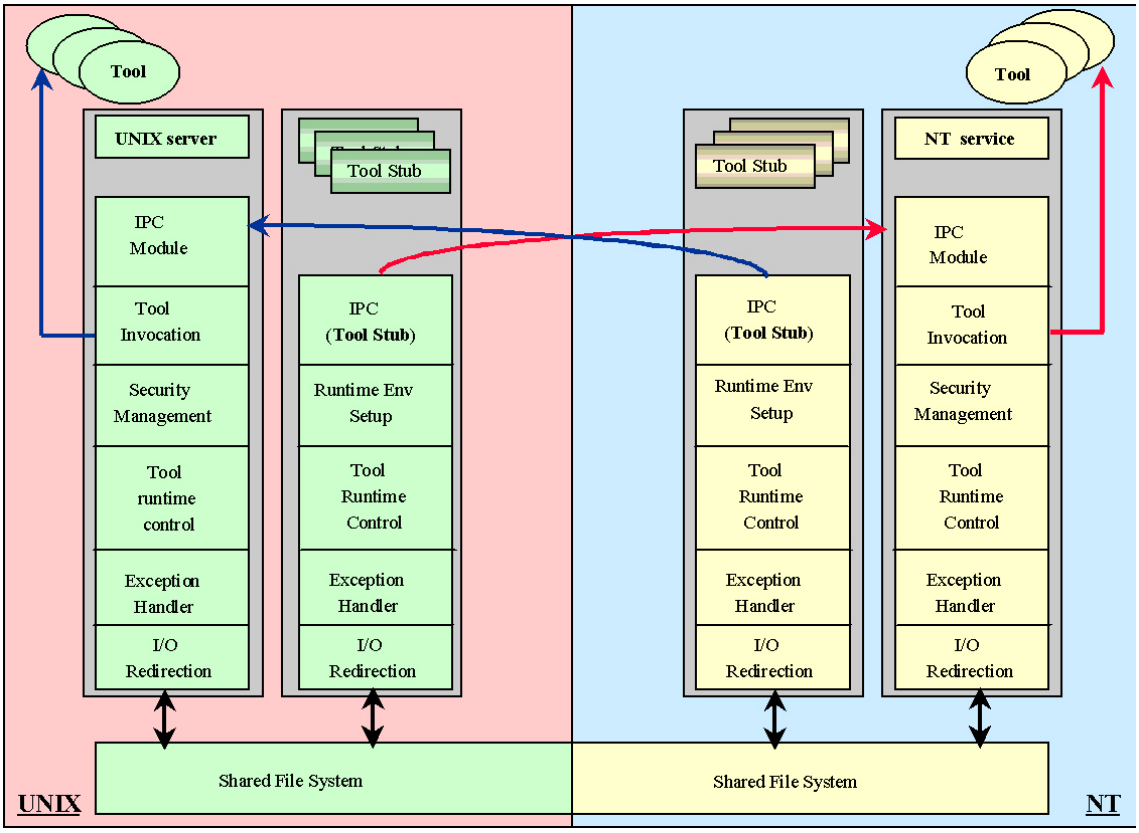

Figure 10: Architecture overview of run-time client-server platform independent execution technology

Conclusion

This paper describes the innovative technical solutions employed by Intel's CAD organization to develop a production capable NT*-UNIX* cross-platform CAD design flow environment for development, maintenance, and deployment activities.

This work is important for Intel to continue to maintain a powerful, effective, and competitive design environment for the future generation of microprocessor design projects. The simple but powerful technique described in this paper is easily proliferable to future Intel® microprocessor design project environments as Intel moves from a UNIX to an IA-NT design flow environment.

Acknowledgments

We thank DT and Computing Technology management for providing the opportunity to work on this exciting project. We are grateful to our department manager Tae Paik for his vision, inspiration, patience, and continuous encouragement during this project. We wish to acknowledge Tae Paik for the brainstorming session that helped us to come up with a run-time client-server model and Daaman Hejmadi who worked with Shesha Krishnapura (author) on the initial concept of a compile-time client-server model. Thanks to Sunil Bhasin and Gil Kleinfeld for applying the ITC Library to new applications. Thanks to Athena-NT Technical Working Group members for ratifying the technical concepts and documents. Special thanks to Tzvi Melamed, Greg Hannon, Chenwei Chiu, and Ming Lin for their technical feedback and encouragement during the development work. Thanks to Srini Rama for implementing parts of the run-time client-server utilities and conducting performance tests. Thanks to Athena and Nike development engineers who are using the compile-time and run-time technologies in their products.

References

[1] Alexander Wolfe, "Intel taps Windows NT in design-software shift." *EETIMES*, issue 948, April 7, 1997, pages 1, 148.

[2] Richard Goering, "Can NT win in IC design?" *EETIMES*, issue 992, page 70.

Authors' Biographies

Shesha Krishnapura is a staff engineer and manager for the Software Platform Engineering Group in CT/DT. He received an M.S. degree in computer science from Oregon State University in 1991 and a B.E. in electronics engineering from the University Vishveshvaraya College of Engineering, India in 1986. He has worked on digital switching system software and CAD platform tools. His interests are in client-server modeling and platform migration. He joined Intel in 1991. His e-mail is shesha.krishnapura@intel.com.

Ty Tang is a senior software engineer in the Software Platform Engineering Group in CT/DT. She received her B.S. in computer science and computer engineering from the University of California at Los Angeles in 1990. Her expertise is in various levels of system programming under UNIX* and NT* platforms, software engineering and migration of CAD tools and system software to various platforms. She joined Intel in 1994. Her e-mail is ty.tang@intel.com.

Vipul Lal is a senior software engineer in the Software Platform Engineering Group in CT/DT. He received a B.S. in computer engineering from the University of Pune, India in 1993. He has developed system and application software for various flavors of UNIX* and NT* operating systems. He joined Intel in 1996 and is now working on cross-platform system and application software development. His e-mail is vipul.lal@intel.com

Formally Verifying IEEE Compliance of Floating-Point Hardware

John O'Leary, Xudong Zhao, Rob Gerth, Carl-Johan H. Seger
Strategic CAD Labs, Intel Corporation, Hillsboro OR

Index words: floating-point, IEEE compliance, formal verification, model checking, theorem proving

Abstract

This paper describes the formal specification and verification of floating-point arithmetic hardware at the level of IEEE Standard 754. Floating-point correctness is a crucial problem: the functionality of Intel's floating-point hardware is architecturally visible (it is documented in the programmer's reference manual [1] as well as an IEEE standard [2]) and, once discovered, floating-point bugs are easily reproduced by the consumer. We have formally specified and verified IEEE-compliance of the Pentium® Pro processor's FADD, FSUB, FMUL, FDIV, FSQRT, and FPREM operations, as well as the correctness of various miscellaneous operations including conversion to and from integers. Compliance was verified against the gate-level descriptions from which the actual silicon is derived and on which all traditional pre-silicon dynamic validation is performed. Our results demonstrate that formal functional verification of gate-level floating-point designs against IEEE-level specifications is both feasible and practical. As far as the authors are aware, this is the first such demonstration.

Introduction

The main objectives of this paper are to describe how we specify IEEE compliance of gate-level designs, and how we employ theorem-proving and model-checking tools to formally verify that the designs meet their specifications. A further objective is to relate our experience verifying the gate-level description of the Pentium® Pro processor's floating-point execution unit.

Work on this project began in July 1997, after the discovery of a post-silicon erratum in the Pentium Pro processor's floating-point execution unit (FEU). The discovery of this erratum was especially disturbing to us because the Pentium Pro FEU had been the subject of a previous formal verification project [3]. The work we describe in this paper extends and improves upon the previous effort in the following respects. First, our current specifications cover numeric correctness at the level of the IEEE standard, while the specifications used in the previous verification project are at the level of functional blocks within the FEU. Second, our verification is much more comprehensive: in the current work we verify correct datapath functionality for all floating-point operations implemented in the FEU, including all precisions, rounding modes, and flags. The focus in the earlier verification was core datapath functionality only. Perhaps the most important improvement over our previous work is methodological: whereas in the previous work we had no metric for the completeness of a property set (leaving the door open for errata to escape), we now employ formal proof of compliance to an IEEE-level specification as our completeness criterion.

Formal verification has also been applied to the AMD-K7* floating-point multiplication, division and square root algorithms [9]. We discuss this and other related work after presenting our results.

Formally verifying IEEE compliance of floating-point hardware presents three major challenges. The first challenge is to capture the sense of the IEEE specification in a straightforward and formal way, such that it can be presented and debated in a verification review. For example, the IEEE standard mandates that "when rounding toward $-\infty$ the result shall be the format's value … closest to and no greater than the infinitely precise result" [2]. By introducing appropriate definitions and abstractions we can state this requirement formally in terms of the rounded result R, infinite precision result V, and the smallest representable increment ulp+:

* All other trademarks are the property of their respective owners.

$$round(toNegInf, R, V) = (R \le V) \wedge (V < R + ulp^{+})$$

Note that writing IEEE-level specifications requires us to be able to specify arithmetic operations of unbounded precision, and that specification notations that support only finite-precision "bit vector" arithmetic are therefore inadequate.

The second challenge arises because the precision of the arithmetic operations implemented by the hardware is inherently bounded. A key aspect of our proof is the verification that the finite-precision computations performed by the hardware correctly implement the unbounded precision operations mandated by the specification. The gap between the finite-precision gate-level description and the unbounded-precision high-level specifications is spanned in the first instance by our word-level model checker, using *hybrid decision diagrams* (HDDs) [4]. At higher levels of abstraction, we provide specialized decision procedures that facilitate reasoning about unbounded precision specifications.

The third challenge is the sheer size and complexity of the floating-point algorithms and the hardware. We have implemented a framework that combines advanced model-checking technology, user-guided theorem-proving software, and decision procedures that facilitate arithmetic reasoning. Our approach has been to develop a lightweight theorem prover that is well suited to hardware reasoning and composing/decomposing model checking results. The result is an environment for developing concise and readable high-level arithmetic proofs. Combining theorem proving and model checking extends the capacity of our verification tools far beyond what is feasible by model checking alone.

This paper is organized as follows. We present a brief overview of our formal verification framework. We then explain in detail the specification and verification of the FMUL instruction to illustrate our methods. The remaining instructions are covered in less detail. FADD/FSUB and miscellaneous operations are verified against algorithmic models, which are verified in turn against IEEE-level specifications using the FMUL methodology. FSQRT, FDIV, and FPREM present special challenges because they are iterative algorithms. We explain how our basic methodology is extended to deal with iterative algorithms. The paper closes with a summary of results and a discussion of related work.

Technology Overview

The work we discuss in this paper was made possible by the formal verification system Forte, currently in development at Intel. Forte is an evolution of the Voss verification system, developed at the University of British Columbia [5]. It seamlessly integrates several types of model-checking engines with lightweight theorem proving and extensive debugging capabilities, creating a productive high-capacity formal verification environment.

There are two model-checking engines in Forte relevant to this work: a checker based on symbolic trajectory evaluation and a word-level model checker. Symbolic trajectory evaluation (STE) is a formal verification method that can be viewed as something of a hybrid between a symbolic model checker and a symbolic simulator [6]. It allows the user to automatically check the validity of a formula in a simple temporal logic but it performs the checking by using a symbolic simulation-based approach and thus is significantly more efficient than more traditional symbolic model-checking approaches. Symbolic trajectory evaluation is particularly well suited to handle datapath properties, and it is used in this work to verify gate-level models against more abstract reference models. However, since symbolic trajectory evaluation is based on binary decision diagrams [7], there are circuit structures that cannot be handled. In particular, multipliers are beyond the capability of STE alone.

The word-level model checker is a linear time logic model checker tailored specifically to verifying arithmetic circuits. By using hybrid decision diagrams, the model checker can handle circuits that BDD-based model checkers cannot handle [4]. In particular, it can handle large multipliers efficiently, satisfying one of the requirements for verifying any modern floating-point unit. Although the model-checking algorithms used by the word-level model checker differ from those used by STE, the word-level model checker uses the STE engine to extract a transition relation from the circuit, ensuring that the two model checkers use a consistent view of the circuit's behavior.

Today, neither model checker is capable of checking the high-level IEEE specifications against the gate-level design. As a result, the verification must be broken up into smaller pieces, either following the structure of the circuit or partitioning the input data space. In order to ensure that no mistakes are made in this partitioning process and that no verification conditions are forgotten, a mechanically checked proof is needed. In Forte, a lightweight theorem proving system is used for this task. This theorem prover is seamlessly and tightly integrated with the model checkers. As a result, no translation or re-formulation of the verification results is needed before the theorem proving can be performed. In addition, some special purpose decision procedures are also integrated, making reasoning about arithmetic results significantly easier and more efficient.

Although the theorem prover in the Forte environment can be used to reason about almost any type of object, its design is tailored specifically to combining model-checking results and reasoning about integer expressions. As a result, our high-level specifications are given in terms of integers, rather than rational numbers. As we show in a later section, this is more of an aesthetic than a real limitation.

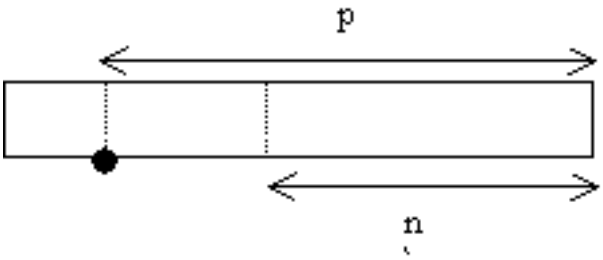

Figure 1: Mantissa representation

Finally, in order to make the verification process practical and efficient, the Forte verification system also includes significant support for efficient debugging. Support ranges from visualization support to provision of extensive counter example generation integrated in all decision procedures.

Specifying IEEE Compliance of Hardware

A floating-point number is made up of a sign $s \in \{-1,1\}$, a mantissa $m \geq 0$, and an exponent $e \geq 0$. Each floating-point number represents the rational number

$$R = \frac{s * m * 2^{e}}{2^{p} * 2^{bias}}$$

where p represents the number of bits in the fractional portion of the mantissa, and is a constant chosen to make e's range non-negative. Figure 1 depicts our representation of the mantissa. For a given R, we define ulp^{+} and ulp^{-} as the distance from R to the next and previous representable values, and ulp as the distance from R to the next largest representable value in magnitude.

$$ulp^{+} = \frac{B^{+}}{2^{p} * 2^{bias}} \quad ulp^{-} = \frac{B^{-}}{2^{p} * 2^{bias}} \quad ulp = \frac{2^{n} * 2^{e}}{2^{p} * 2^{bias}}$$

For most values of R, $B^{+} = B^{-} = 2^{n} * 2^{e}$ where p-n is the number of significant bits in the fractional part of the mantissa (p - n = 23 for single precision, 52 for double precision, and 64 for extended precision). When R's mantissa is 1.0, the adjacent representable values on either side of R have different exponents. In such cases $B^{+} = 2^{n} * 2^{e-1}$ and $B^{-} = 2^{n} * 2^{e}$ when R is negative; and $B^{+} = 2^{n} * 2^{e}$ and $B^{-} = 2^{n} * 2^{e-1}$ when R is positive. The bottom n bits in the mantissa are all 0.

The IEEE standard specifies that the result of a floating-point operation is computed as if the operation has been performed to unbounded precision and then rounded to fit into the destination format. The result of rounding a floating-point number is one of the two representable values closest to the exact value; which one of the closest values is determined based on the rounding mode. The IEEE standard specifies four rounding modes: toward zero, toward negative infinity, toward positive infinity, and to nearest. Since the IEEE standard is written in English, the formalization we present in this paper is necessarily our interpretation of the standard's intent.

We capture the required relationship between a result V calculated to unbounded precision and a result R rounded toward negative infinity

$$\text{round}(\text{toNegInf}, R, V) = (R \leq V) \wedge (V < R + ulp^{+})$$

The relation between a result V calculated to unbounded precision and a rounded result R is specified as follows for each of the three remaining rounding modes:

$$\text{round}(\text{toPosInf}, R, V) = (R - ulp^{-} < V) \wedge (V \leq R)$$

$$\text{round}(\text{toZero}, R, V) = (|R| \leq |V|) \wedge (|V| < |R| + ulp)$$

$$\text{round}(\text{toNearest}, R, V) = (R - \tfrac{1}{2} ulp^{-} \leq V) \wedge (V \leq R + \tfrac{1}{2} ulp^{+}) \wedge ((R - \tfrac{1}{2} ulp^{-} = V) \vee (V = R + \tfrac{1}{2} ulp^{+}) \Rightarrow \text{even}(R))$$

Even(R) means that the least significant bit in R's mantissa is 0, and it depends on the destination precision of the floating-point operation.

Specifying FMUL

To illustrate our methods, we will derive the specification for floating-point multiplication.

With the definitions of the previous section in hand, capturing the IEEE-level specification is easy. The product of two floating-point quantities, to unbounded precision, is

$$V_{FMUL} = \frac{s_1 * m_1 * 2^{e_1}}{2^{p} * 2^{bias}} * \frac{s_2 * m_2 * 2^{e_2}}{2^{p} * 2^{bias}}$$

Let us call the result of the computation

$$R_{FMUL} = \frac{s * m * 2^e}{2^p * 2^{bias}}$$

The units in the last place are as defined in the previous section, and the IEEE-level specification is

$$round(rndMode, R_{FMUL}, V_{FMUL})$$

where $rndMode \in \{toZero, toNegInf, toPosInf, toNearest\}$.

Since our verification environment only deals with specifications written in terms of integer operations, we now need to transform the IEEE-level specification developed above into an equivalent specification involving only operations on integers. Expanding the definitions of V, R, and ulp^+ in the IEEE specification for multiplication rounding to $-\infty$ gives

$$round(toNegInf, R_{FMUL}, V_{FMUL}) =$$
$$\left(\frac{s*m*2^e}{2^p*2^{bias}} \le \frac{s_1*m_1*2^{e_1}}{2^p*2^{bias}} * \frac{s_2*m_2*2^{e_2}}{2^p*2^{bias}}\right) \wedge$$
$$\left(\frac{s_1*m_1*2^{e_1}}{2^p*2^{bias}} * \frac{s_2*m_2*2^{e_2}}{2^p*2^{bias}} < \frac{s*m*2^e}{2^p*2^{bias}} + \frac{B^+}{2^p*2^{bias}}\right)$$

Now, by multiplying both sides of each inequality by $(2^p * 2^{bias})^2$ and rearranging, we obtain an equivalent specification in terms of integer operations only:

$$round(toNegInf, R_{FMUL}, V_{FMUL}) =$$
$$\left((s*m*2^e)*2^{p+bias} \le V'\right) \wedge$$
$$\left(V' < (s*m*2^e + B^+)*2^{p+bias}\right)$$

where $V' = (s_1 * m_1 * 2^{e_1}) * (s_2 * m_2 * 2^{e_2})$.

For the other rounding modes, we follow a similar procedure and obtain the following specifications:

$$round(toPosInf, R_{FMUL}, V_{FMUL}) =$$
$$\left((s*m*2^e - B^-)*2^{p+bias} < V'\right) \wedge$$
$$\left(V' \le (s*m*2^e)*2^{p+bias}\right)$$
$$round(toZero, R_{FMUL}, V_{FMUL}) =$$
$$\left(\left|s*m*2^e\right|*2^{p+bias} \le |V'|\right) \wedge$$
$$\left(|V'| < \left(\left|s*m*2^e\right| + 2^n * 2^e\right)*2^{p+bias}\right)$$
$$round(toNearest, R_{FMUL}, V_{FMUL}) =$$
$$\left((s*m*2^e - \tfrac{1}{2}B^-)*2^{p+bias} \le V'\right) \wedge$$
$$\left(V' \le (s*m*2^e + \tfrac{1}{2}B^+)*2^{p+bias}\right) \wedge$$
$$\left(\begin{array}{l}((s*m*2^e - \frac{1}{2}B^-)*2^{p+bias} = V') \vee \\ \quad (V' = (s*m*2^e + \frac{1}{2}B^+)*2^{p+bias}) \\ \quad \Rightarrow (m \% 2^n = 0)\end{array}\right)$$

These specifications are all in terms of the integer operations addition, subtraction, multiplication, and modulus and are therefore amenable to word-level model checking.

Verifying FMUL

The IEEE-level specifications are much too large and abstract to be directly verified by model checking, so the first step in our verification is to decompose the specification into smaller, more concrete pieces. The decomposition is done manually using engineering judgement about what is feasible to verify with the model checker. The second step is to verify the lower-level specifications by model checking. Typically, several iterations through the decomposition and model-checking steps are required. The third step is to formally compose the lower-level specifications using the theorem prover. The composition should yield back the desired high-level specification, thus verifying the correctness of the decomposition step.

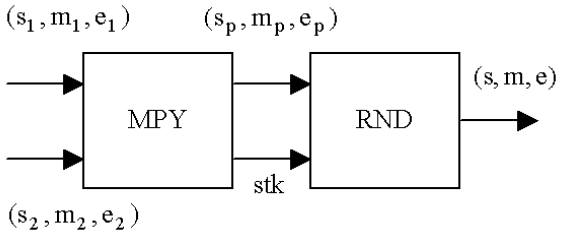

Figure 2: Floating point multiplier

Figure 2 shows a block diagram of the basic floating-point multiplication algorithm. In the MPY stage, inputs (s_1, m_1, e_1) and (s_2, m_2, e_2) are multiplied to compute an approximate result (s_p, m_p, e_p) without regard to rounding. Algorithmically, this means that their mantissas are multiplied, their exponents are added, and their sign bits are XORed. The output of the multiplication stage is a truncated form of the unbounded-precision result. A sticky bit is also generated, which is set if any significance was lost in the truncation. The intermediate result and sticky bit are sent to the rounder, which rounds according to the rounding mode and destination format.

The verification task was broken into multiplication and rounding stages, following the natural structure of the hardware. The multiplication stage was further decomposed into operations on the mantissa, exponent, and sign. Properties at this level were verified by model checking, and the results combined to yield the required high-level result.

First consider the multiplication stage. Using word-

level model checking, we verify the following properties of the mantissa, exponent, and sign computations:

$$m_p \le m_1 * m_2$$
$$m_p + 1 > m_1 * m_2$$
$$e_p = e_1 + e_2 + p + \text{bias}$$
$$s_p = s_1 * s_2$$

We also verify that the sticky bit correctly captures the loss of precision in multiplication.

$$stk = (m_p < m_1 * m_2)$$

We combine the above results using the theorem prover to obtain

$$\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| \le \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|$$
$$\left|s_p * (m_p + 1) * 2^{e_p + p + \text{bias}}\right| > \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|$$
$$stk = \left(\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| < \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|\right)$$

This completes verification of the multiplication part of FMUL.

Verification of the rounding property, like the property of multiplication, required proving some lower-level properties using word-level model checking and manipulating them using the theorem prover. For the rounding calculation, we verified the high-level property

$$\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| \le \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right| \wedge$$
$$\left|s_p * (m_p + 1) * 2^{e_p + p + \text{bias}}\right| > \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right| \wedge$$
$$stk = \left(\left|s_p * m_p * 2^{e_p + p + \text{bias}}\right| < \left|s_1 * m_1 * 2^{e_1} * s_2 * m_2 * 2^{e_2}\right|\right) \text{ for}$$
$$\Rightarrow$$
$$\text{round}(\text{rndMode}, R, V)$$
$$\text{rndMode} \in \{\text{toZero}, \text{toNegInf}, \text{toPosInf}, \text{toNearest}\}.$$

To complete the verification, we compose the properties of the multiplier and rounder to yield the required high-level property. This composition is performed in the theorem prover, using the logical rule that states "if we have proven both A and $A \Rightarrow B$, , then we can conclude B." In this case A is the property proved of the multiplier (which is also the antecedent of the rounder property) and $B = \text{round}(\text{rndMode}, R, V)$ is the desired property of the multiplier combined with the rounder. It is clear that B follows immediately.

Verifying FADD/FSUB and Miscellaneous Operations

The verification strategy for FMUL was a *structural decomposition* driven by the structure of the gate-level design and by capacity limitations of the model checker. For a large class of operations, which includes floating-point add and subtract, normalize-and-round, conversion to and from integers, and various shift instructions, model checking capacity suffices to perform *black box* verification. The strategy deployed here is to build an abstract *reference model* against which the gate-level design is verified. In a separate step, we can verify the reference model against IEEE-level properties[1]. As an example, Figure 3 shows the reference model for an operation used internally by the Pentium® Pro processor. Note that this is an algorithmic description that takes two Boolean vectors (S1 and S2) and returns a Boolean vector with an appropriately shifted mantissa.

```
let SHR_SPEC S1 S2 =
  let diff =
    exp S1 '<' exp S2 => exp S2 '-' exp S1
    | exp S1 '-' exp S2 in
  let shiftv = diff '< max => diff | max in
  let man = shr (man S2) shiftv in
  let exp = man '= 0 => 0 | exp S2 in
  (sgn S2)@exp@man;
```

Figure 3: Reference model for FPSHR

The IEEE-level specification of FADD/FSUB is similar to that for FMUL. For FADD/FSUB, we used as a reference model a textbook algorithm for addition/subtraction with rounding [8]. The gate-level design was verified against the reference model using symbolic trajectory evaluation. The verification of the gate-level design against the reference model is black box in the sense that no decomposition is needed of the reference model or the design. The reference model (but not the gate-level model) performs addition in five stages: mantissa alignment (so that the exponents are equal), addition/subtraction of the mantissas, normalization, rounding, and renormalization (as rounding may produce an overflow). The proof that the reference model complies to IEEE specifications naturally decomposes into the same stages. For each stage, the appropriate properties were proved (using word-level model checking), and were finally combined using the

[1] Many of the miscellaneous operations are used internally by the processor and lack meaningful IEEE-level specifications. For these operations the second step is unnecessary.

Forte theorem prover.

While the verification of the reference model against the high-level specification required structural decomposition, the decomposition was determined by what was most convenient for theorem proving and not by constraints imposed by the gate-level design or the model checker. The advantage of this approach is that the most time-consuming and complex part of the verification, the use of theorem proving in verifying the high-level specification, is completely isolated from the relatively fast-changing gate-level design. As a case in point, we ported the FADD/FSUB reference models developed for the Pentium Pro processor to two other processors and verified them with only very minor changes that did not impact the high-level proofs. The high-level proofs therefore remain valid for the other processors.

Verifying FSQRT and FDIV

Division, square root, and remainder present special challenges. They are both more difficult to specify and more difficult to verify than FMUL and FADD/FSUB.

The difficulty in specification is minor, and it arises because our verification framework requires that specifications be stated in terms of integer operations. However, FDIV naturally yields a rational result, and FSQRT may well yield an irrational result. Following the method used for FMUL, we define

$$V_{FDIV} = \frac{s_2 * m_2 * 2^{e_2}}{2^p * 2^{bias}} \Big/ \frac{s_1 * m_1 * 2^{e_1}}{2^p * 2^{bias}}$$

$$R_{FDIV} = \frac{s * m * 2^e}{2^p * 2^{bias}}$$

Simple algebra gives

$$V_{FDIV} = \frac{s_2 * m_2 * 2^{e_2}}{s_1 * m_1 * 2^{e_1}}$$

Since $s_1, s_2 \in \{-1,1\}$, it is true that $s_2 / s_1 = s_1 * s_2$. Therefore the above definition can be rewritten as follows (the utility of this step will be explained in a moment):

$$V_{FDIV} = \frac{s_1 * s_2 * m_2 * 2^{e_2}}{m_1 * 2^{e_1}}$$

We can then derive the following specification, in terms of integer operations, for division rounding to negative infinity:

$$\begin{aligned} &\text{round}(\text{toNegInf}, R_{FDIV}, V_{FDIV}) = \\ &\quad \left(s * m * 2^e * m_1 * 2^{e_1} \le s_1 * s_2 * m_2 * 2^{e_2} * 2^{p+bias}\right) \wedge \\ &\quad \left(s_1 * s_2 * m_2 * 2^{e_2} * 2^{p+bias} < \left(s * m * 2^e + B^+\right) * m_1 * 2^{e_1}\right) \end{aligned}$$

Our derivation involved multiplying both sides of each inequality by $m_1 * 2^{e_1}$. If we had multiplied by $s_1 * m_1 * 2^{e_1}$ we would have had to break our final specification into cases according to the sign of s_1. Rearranging V_{FDIV} using the fact $s_2 / s_1 = s_1 * s_2$ allows our final specification to remain simple. Specifications for division in other rounding modes are also easy to derive.

For FSQRT, we define

$$V_{FSQRT} = \sqrt{\frac{s_2 * m_2 * 2^{e_2}}{2^p * 2^{bias}}}$$

$$R_{FSQRT} = \frac{s * m * 2^e}{2^p * 2^{bias}}$$

We derive the specifications for square root by squaring both sides of each inequality and rearranging them. For example, the specification for square root rounding to negative infinity is

$$\begin{aligned} &\text{round}(\text{toNegInf}, R_{FSQRT}, V_{FSQRT}) = \\ &\quad \left(\left(s * m * 2^e\right)^2 \le \left(s_2 * m_2 * 2^{e_2}\right) * 2^{p+bias}\right) \wedge \\ &\quad \left(\left(s_2 * m_2 * 2^{e_2}\right) * 2^{p+bias} < \left(s * m * 2^e + B^+\right)^2\right) \end{aligned}$$

The FPREM (floating-point remainder) instruction is a variant of FDIV and will not be discussed further.

FDIV, FSQRT and FPREM are realized in the Pentium® Pro processor by iterative algorithms, and the complexity of the algorithms makes them difficult to verify. Verification of these operations required verification of a loop invariant plus some special properties of the initial and final iterations. These results were then combined with properties of the rounding algorithm.

First, we need to verify the correctness of the control flow, shown in Figure 4. Suppose the computation takes n iterations. We must prove that the circuit is in its initial state in cycle 0, in its loop state at cycles 1 through n, in the final state of the FDIV or FSQRT computation in cycle n+1, and rounding the result in cycle n+2. These properties are verified by induction over time. The induction base says that at cycle 0, the circuit is in the initial state, the loop counter is n, and

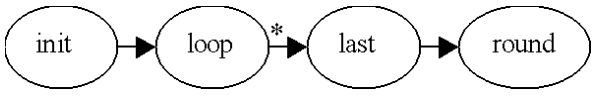

Figure 4: FDIV/FSQRT/FPREM algorithm flow

the next state is the loop state. The induction step says if the circuit is in the loop state and the loop counter is greater than 0, then the next state is the loop state and the counter decreases by 1. Furthermore, if the circuit is in the loop state and the loop counter is 0, then the next state is the final state, and the result will be rounded after the final state. All of the induction base and induction steps are proved by model checking on the circuit. The theorem prover is used to compose these results to verify the correctness of the control flow.

The second step is to prove the data invariants. There are two data invariants for the iterative operations. The first invariant is the range invariant, which relates the values of the partial remainder and divisor (or radicand) in an iteration. The second invariant is the value invariant, which relates the values of the partial remainder, divisor, dividend, and quotient (or partial remainder, radicand, and root) in adjacent iterations. The range and value invariants are proved using induction over time. The base cases and the induction steps are proved by model checking. The theorem prover is used to combine base and induction steps to complete the invariant proofs.

After the second step is done, the results are combined with properties of the rounding stage to reach the required high-level specification. This verification step is very similar to the final verification step for FMUL.

Results on the Pentium® Pro Processor

The preceding sections have given an overview of our verification methodology and its application to various classes of operations. In this section we describe the results of verifying the Pentium® Pro processor's floating-point execution unit (FEU).

This work has its roots in earlier Intel work on arithmetic circuit verification, which focused on specifying and verifying core datapath functionality [3]. Our goals for the current work were much broader in scope. Our aims for the specification and verification effort were as follows:

1. Construct IEEE-level functional specifications for all floating-point operations performed by the FEU.
2. Verify correct datapath functionality for all operations specified.
3. Verify that flags are correctly set and faults are correctly signaled, as required by the microarchitecture specification.
4. Cover all variants of each operation, including those used only by microcode.
5. Cover all precisions and rounding modes.

We accomplished these aims during a five-quarter project involving two engineers full-time, with one additional full-time engineer involved in the fifth quarter. The first two quarters were devoted mostly to tool and methodology development, driven by example operations drawn from the Pentium Pro processor. In this initial period, our word-level model checker, theorem-proving software, and decision procedures were refined, and inference rules were developed to reason about properties proved by model checking. In the final three quarters, our resources were applied to verification in earnest, with work on tools and infrastructure only as needed. In all, we undertook six verification sub-tasks: FADD/FSUB (we consider this one operation), FMUL, FDIV, FSQRT, FPREM, and numerous miscellaneous operations (each is relatively simple, but there are many of them). Each sub-task took approximately one engineer-quarter[2], including specification development, understanding the FEU micro architecture and gate-level design, model checking, and theorem proving.

To limit the scope of the project we chose to focus on the numeric correctness of the FEU. Some important correctness properties are not covered by our verification, including freedom from interference or contention between multiple active operations, the correct response of the FEU to external control events (stalls, for example), and the correctness of the microcode that uses the FEU operations.

A further goal of our Pentium Pro processor verification effort was that our results and methodologies should be reusable within and between major processor generations. To demonstrate the potential for such reusability, we repeated some of our verifications on other Intel® processors. We repeated the model checking of FADD/FSUB and miscellaneous operations on a proliferation of the Pentium Pro processor, and we verified that the proliferation's behavior was identical with respect to our specifications (despite some redesign that had occurred). We observed virtually complete reuse of our properties and verifica-

[1]This figure includes time for staff meetings and other routine activities.

tion scripts. We repeated verification of FADD/FSUB on a new processor generation that has a completely different microarchitecture and, again, obtained extremely high re-use of properties and scripts. The highly reusable nature of the FADD/FSUB verification is a consequence of our capability to perform black box model checking on these operations. We also ported our FMUL verification to the new processor. Since FMUL was verified using a structural decomposition approach, microarchitectural changes between processor generations dictated that many of the low-level properties used in the FMUL verification had to be modified. Our tools and methodologies proved fully reusable, however. One very high-quality erratum was discovered and corrected in the pre-silicon design phase.

Discussion

The success of this project has three principal lessons. The first is that we can formally verify gate-level descriptions of floating-point hardware against IEEE-level specifications using the Forte formal verification framework. Moreover, it is feasible and practical to do so in the timeframe of a major processor design project. The major impact of our work is on the quality of the product: our method will eliminate a class of post-silicon escapes due to incorrect floating-point functionality. A second impact is on validation efficiency: scarce simulation cycles can be redirected to cover functionality that is not covered by formal verification.

The second lesson is that verifying floating-point hardware against IEEE-level specifications requires significant effort and investment. Our verification effort on the Pentium® Pro processor involved two formal verification experts full-time over five quarters, with a third full-time expert added in the final quarter. It is true that part of this time was spent on tool and methodology development, but it is also true that we were able to leverage prior knowledge about the Pentium Pro microarchitecture. We expect that an effort on a future processor will require a similar investment. On the other hand, the payback is potentially very high. Each floating-point erratum has the potential to be a highly visible flaw in the processor, as was the case with the Pentium® processor's FDIV flaw.

A third lesson is that devising and executing floating-point formal verification strategies requires certain specialized expertise. Identifying low-level properties or writing reference models requires solid knowledge of floating-point microarchitecture and algorithms. Relating the low-level properties to IEEE-level specifications requires skill in constructing formal proofs, and mechanizing the verification to the extent described in this paper requires skill and experience in using theorem-proving software. The difficulties encountered in model checking range from moderate (FADD/FSUB and miscellaneous operations) to very challenging (FMUL). While the required expertise can be readily taught and learned, it is not yet a part of the mainstream curriculum in computer science or electrical engineering.

Most formal verification methodologies in use within industry today focus on using model checkers to verify low-level properties of hardware. A notable exception in the field of floating-point verification is the work of Russinoff on verifying the AMD-K7 * algorithms using the ACL2 theorem prover [9]. Russinoff formally verified the behavior of abstract models of the AMD-K7 floating-point algorithms. Russinoff's abstract models were "derived from an executable model that was written in C and used for preliminary testing" and assumed integer addition, multiplication, and bitwise logical operations as primitives [9]. Russinoff does not specify how he validates the correctness of the abstract models with respect to the gate-level design. In contrast, we verified the IEEE-compliance of the gate-level design from which the silicon is derived and which is used for all pre-silicon dynamic validation. Another important difference between Russinoff's work and ours is that the AMD-K7 verification was carried out in a general-purpose theorem prover whereas our verification methodology combines model checking with a theorem prover that is specialized to manipulate model checking results.

A precursor to the current work is that of Aagaard and Seger, who formally verified the IEEE compliance of the gate-level implementation of a floating-point multiplier using a combination of theorem proving and model checking [10]. The verification we describe in the current paper covers many more operations, and we have demonstrated that our work scales to industrial-sized floating-point hardware.

There has been substantial work on formal verification of floating-point algorithms implemented in either microcode or software. The microcode that implements the floating-point divide instruction of the AMD-K5* processor was verified by Moore, Lynch, and Kaufmann [11]. Russinoff verified the microcode for the AMD-K5 floating-point square root instruction [12]. Harrison describes the specification and verification of the exponential function [13]. Cornea-Hasegan describes the verification of programs that compute IEEE-compliant division, remainder, and

* All other trademarks are the property of their respective owners.

square root operations [14]. All these verifications assume, as a starting point, that the underlying hardware correctly implements arithmetic building blocks such as addition, subtraction, and multiplication. Our work is complementary in that it focuses on verifying the correctness of the underlying hardware. The algorithm verifications have often relied on relatively deep mathematics, and therefore are most appropriately formalized in a general-purpose theorem-proving environment.

Conclusion

We have developed tools and methods for formal verification of floating-point functionality and demonstrated them on the Pentium® Pro processor and other processors. No other formal verification method has been shown to be able to fully span the semantic gap between gate-level descriptions and high-level IEEE specifications in this domain.

Acknowledgments

We are grateful to John Harrison for pointing out an error in an earlier version of our specifications. We thank Yirng-An Chen for helping in the development of the word-level model checker and some early verification work, and Mark Aagaard and Donald Syme for helping in development of the theorem prover. Tom Melham and Robert Jones made key contributions to the methodology we used to verify FADD/FSUB and miscellaneous operations. We are grateful to Timothy Kam, Yatin Hoskote, and Pei-Hsin Ho for their contributions to our earlier verification effort.

References

[1] Intel® Architecture Software Developer's Manual. Volume 2: Instruction Set Reference, Intel Corporation, 1997, Order Number 243191.

[2] "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.

[3] Y-A. Chen, E. Clarke, P-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking," in M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design*, Volume 1166 of *Lecture Notes in Computer Science,* Springer-Verlag, 1996.

[4] E.M. Clarke, M. Khaira, and X. Zhao, "Word-level symbolic model checking: a new approach for verifying arithmetic circuits," in *Proceedings of the 33rd ACM/IEEE Design Automation Conference,* IEEE Computer Society Press, June 1996.

[5] C. Seger, *Voss—A Formal Hardware Verification System: User's Guide,* Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.

[6] C. Seger and R. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design,* 6(2), April 1994, pp. 147-189.

[7] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers,* 35(8), August 1986, pp. 677-691.

[8] J. Feldman and C. Retter, *Computer Architecture: A Designer's Text Based on a Generic RISC*, McGraw-Hill, 1994.

[9] D. Russinoff, "A mechanically checked proof of IEEE compliance of the floating-point multiplication, division, and square root algorithms of the AMD-K7* processor," *London Mathematical Society Journal of Computation and Mathematics,* 1, December 1998, pp. 148-200.

[10] M. Aagaard and C. Seger, "The formal verification of a pipelined double-precision IEEE floating-point multiplier," in *International Conference on Computer-Aided Design,* IEEE Computer Society Press, November 1995.

[11] J. Moore, T. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5 86* floating-point division program," *IEEE Transactions on Computers,* 47(9), September 1998, pp. 913-926.

[12] D. Russinoff, "A mechanically checked proof of correctness of the AMD-K5* floating point square root microcode," *Formal Methods in System Design,* 14(1), 1999, special issue on arithmetic circuits.

[13] J. Harrison, "Floating point verification in HOL Light: the exponential function," Technical Report 428, University of Cambridge Computer Laboratory, June 1997.

[14] M. Cornea-Hasegan, "Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms," *Intel Technology Journal,* Q2 1998 at http://developer.intel.com/technology/itj/q21-998.htm.

Authors' Biographies

John O'Leary is a senior engineer with Strategic CAD Labs, Intel Corporation, Hillsboro, OR. From 1987 to 1990, he was a member of the scientific staff at Bell-Northern Research, Ottawa, Canada. He joined Intel in 1995 after earning a Ph.D. in electrical engineering from Cornell University. His interests are formal hardware specification and verification. His e-mail is joleary@ichips.intel.com.

Xudong Zhao is a staff engineer with Strategic CAD Labs, Intel Corporation, Hillsboro, OR. He recieved his Ph.D. degree in computer science from Carnegie Mellon University in 1996. His research topics include formal verification for hardware, for arithmetic circuits in particular. His e-mail is xzhao@ichips.intel.com.

Rob Gerth is a staff engineer in the Strategic CAD Laboratories, Intel Corporation, Hillsboro, OR. He received his Ph.D in computer science from Utrecht University, The Netherlands in 1989 and was a lecturer at Eindhoven University of Technology before he joined Intel in 1997. His current interests include multi-processor verification. His e-mail is robgerth@ichips.intel.com.

Carl-Johan H. Seger received his Ph.D. degree in computer science from the University of Waterloo, Canada, in 1988. After two years as Research Associate at Carnegie Mellon University he became an Assistant Professor in the Department of Computer Science at the University of British Columbia, and in 1995 he became Associate Professor. He joined Intel's Strategic CAD Labs in 1995. His research interests are formal hardware verification and asynchronous circuits. He is the author of the Voss hardware verification system and is co-author of *Asynchronous Circuits* (Springer-Verlag, 1995). His e-mail is cseger@ichips.intel.com.

Defect-Based Test: A Key Enabler for Successful Migration to Structural Test

Sanjay Sengupta, MPG Test Technology, Intel Corp.
Sandip Kundu, MPG Test Technology, Intel Corp.
Sreejit Chakravarty, MPG Test Technology, Intel Corp.
Praveen Parvathala, MPG Test Technology, Intel Corp.
Rajesh Galivanche, MPG Test Technology, Intel Corp.
George Kosonocky, MPG Test Technology, Intel Corp.
Mike Rodgers, MPG Test Technology, Intel Corp.
TM Mak, MPG Test Technology, Intel Corp.

Index words: structural test, functional test, ATE, DPM, logic test, I/O test, cache test, AC loopback test, inductive fault analysis, fault models, stuck-at fault, bridge fault, delay fault, open fault, defect-based test, ATPG, fault simulation, fault modeling, DPM, test quality, fault grading, design-for-test

Abstract

Intel's traditional microprocessor test methodology, based on manually generated functional tests that are applied at speed using functional testers, is facing serious challenges due to the rising cost of manual test generation and the increasing cost of high-speed testers. If current trends continue, the cost of testing a device could exceed the cost of manufacturing it. We therefore need to rely more on automatic test-pattern generation (ATPG) and low-cost structural testers.

The move to structural testers, the new failure mechanisms of deep sub-micron process technologies, the raw speed of devices and circuits, and the compressed time to quality requirements of products with shorter lifecycles and steeper production ramps are adding to the challenges of meeting our yield and DPM goals. To meet these challenges, we propose augmenting the structural testing paradigm with defect-based test.

This paper discusses the challenges that are forcing us to change our testing paradigm, the challenges in testing the I/O, cache and logic portions of today's microprocessors, due to the paradigm shift, and the problems to be solved to automate the entire process to the extent possible.

Introduction

Traditionally, Intel has relied on at-speed functional testing for microprocessors as this kind of test has historically provided several advantages to screen defects in a cost-effective manner. Unlike other test methods, functional testing does not require the behavior of the device under test (DUT) to be changed during the test mode. Thus, functional testing allows us to test a very large number of "actual functional paths" at speed using millions of vectors in a few milliseconds; to thoroughly test all device I/Os with "tester-per-pin" ATE technology; and to test embedded caches in a proper functional mode. In addition, the testing is done in a noise environment comparable to system operation. However, functional testing is facing an increasing number of obstacles, forcing Intel to look at alternative approaches.

We begin this paper by describing the problem of continuing with functional testing of microprocessors. We then define an alternative paradigm, which we call structural test. Finally, the challenges that we face and the problems that need to be solved to test the logic, I/O, and cache subsystems of the microprocessor to make the alternative test method work are discussed.

Structural testing has been in use in the industry for quite some time. In order to meet Intel's aggressive yield and DPM goals, we propose enhancing the structural test flow, by using defect-based test (DBT). DBT is based on generating manufacturing tests that target actual physical defects via realistic fault models. The primary motivation in augmenting structural testing with DBT is to make up for some of the potential quality losses in migration to structural test methods as well as to meet the challenges of sub-micron defect behavior on the latest high-performance microprocessor circuits. Although the impact of DBT on defects per million products shipped is not well characterized, prelimi-

nary studies of DBT [1] show that it improves quality.

DBT requires a whole suite of CAD tools for its successful application. In section 5, we discuss tool requirements for successful DBT for the latest high-performance microprocessors.

The Microprocessor Test Problem

Stated simply, the increasing cost of testing microprocessors to deliver acceptable product quality on ever faster and more complex designs is the main problem we face. The cost challenges range from the non-recurring design and product engineering investment to generate good quality tests to the capital investment for manufacturing equipment for test.

Automatic Test Equipment (ATE) Cost

Following Moore's Law for the past two decades, the silicon die cost of integrated circuits has decreased as the number of transistors per die has continued to increase. In contrast, during the same period, the cost of testing integrated circuits in high-volume manufacturing has been steadily increasing. Silicon Industry Association (SIA) forecasts, depicted in Figure 1, predict that the cost of testing transistors will actually surpass the cost of fabricating them within the next two decades [2].

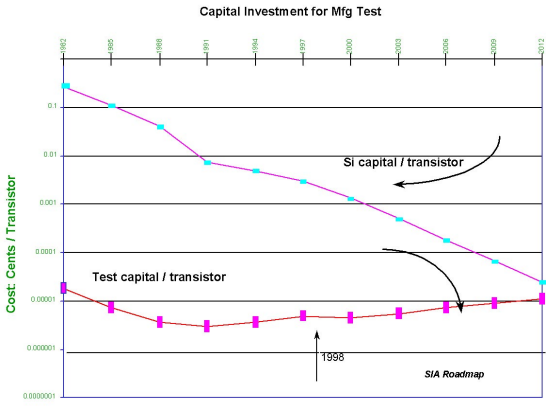

Figure 1: Fabrication and test cost trends

Lagging ATE Technology

Aggressive performance targets of Intel's chip set and microprocessor products also require increasingly higher bus bandwidth. Due to problems such as power supply regulation, temperature variation, and electrical parasitics, tester timing inaccuracies continue to rise as a function of the shrinking clock periods of high-performance designs. The graph in Figure 2 shows trends for device period, overall tester timing accuracy (OTA), and the resulting percentage yield loss. It was derived from information in the SIA roadmap.

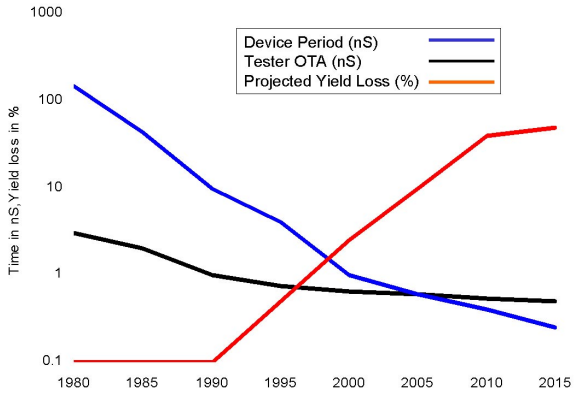

Figure 2: Tester accuracy and projected yield loss trends

In addition to the increase in device frequency and the number of I/O pins, advanced signaling techniques are also used to improve I/O performance. One such signaling innovation is the use of the source-synchronous bus, which has been in use since the Pentium® Pro line of microprocessors. Data on such a bus is sent along with a clock (strobe) generated from the driving device. This complicates testing since the ATE needs added capability to synchronize with the bus clock (strobe).

Test Generation Effort

Manual test writing, which has been in use at Intel, requires a good understanding of the structure of the DUT (typically a design block owned by a designer), as well as global knowledge of the micro-architecture. The latter is required since tests have to be fed to the DUT and the response read from the DUT. With increasing architectural complexities such as deep pipelining and speculative execution, increasing circuit design complexity and new failure modes, the cost of test writing is expected to become unacceptable if we are to meet time-to-volume targets. This is supported by the data presented in Figure 3 where manual test generation effort is compared with the effort required if ATPG, augmented with some manual test generation, were used. Note that manual test writing effort required for functional testing has been increasing exponentially over the last several generations of microprocessors at Intel. Compared to that, the projection for ATPG is very small. Note that the data for Willamette/Merced™ and beyond are projections.

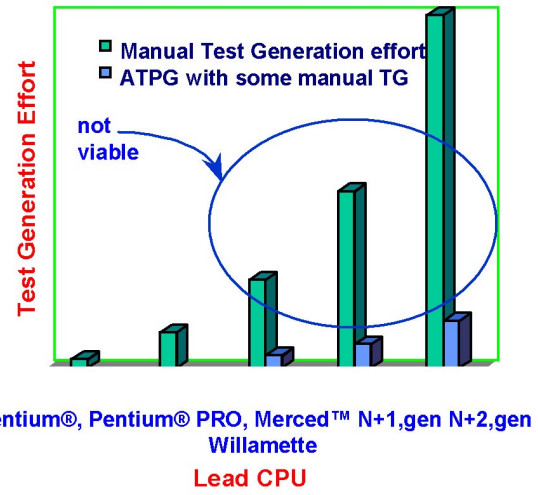

Figure 3: Test generation effort trend

Deep Sub-Micron Trends

As the feature length of transistors scales down, power supply voltage is scaled down with it, thereby reducing noise tolerance. Metal pitch is also scaled in tandem to realize the density gain. If interconnects were scaled proportionately in both pitch and height, line resistivity would rise quadratically, thereby degrading performance. To hold this trend down, metal height is scaled down by a smaller factor than the pitch, which results in increased cross capacitance.

The increase in the number of metal layers introduces more masking steps and can skew the random defect distribution towards interconnect failure modes such as bridges and open vias. Susceptibility to process variation is heightened due to the higher cross capacitance and reduced noise tolerance.

Like other CAD tools, performance validation tools are struggling to keep up with increasing design sizes and circuit design complexity. The most common solution is to build simplifying assumptions into the tools, and to offset this by the use of conservative nominal delays. Faced with increasing performance goals, designers build devices with negative timing margins. Such aggressive designs styles, coupled with increasing layout density, mean that even minor defects or process variations, which would otherwise be benign, could result in failures. Deliberate design marginality thus translates into test problems. Writing functional tests for subtle failure modes, which are made manifest under a very specific set of conditions, is becoming increasingly difficult.

Test Paradigm Shift and Challenges of the New Test Paradigm

Test paradigms are defined by (a) the *kind of test*; (b) the *kind of tester* that stores and delivers the test; and (c) the test *delivery mechanisms*.

Tests can be either *structural* or *functional*. Structural tests target manufacturing defects and attempt to ensure the manufacturing correctness of basic devices such as wires, transistors, etc. Functional tests, on the other hand, target device functionality and attempt to ensure that the device is functioning correctly. Functional tests are written primarily for architectural verification and silicon debug. They can be used for manufacturing testing also, as is done at Intel. Structural tests, on the other hand, are used primarily for manufacturing testing.

Testers come in two varieties: *functional* and *structural*. Functional testers can drive a large number of I/O pins at high clock rates with great timing accuracy. On the other hand, structural testers are limited in the number of I/O pins they can drive, as well as the speed and accuracy with which they can deliver data to the I/O pins. The cost of structural testers is considerably lower than the cost of functional testers.

Tests can be delivered in one of two ways. The device's normal functional channels are used and the device runs at operating speed. Alternatively, special design-for-test (DFT) channels can be designed, and tests are applied through these channels at less than operational speed. The scan structure and ArrayDAT exemplify this.

The test paradigm in use at Intel so far uses functional testers and functional tests. These tests are delivered using the functional channels. Functional tests are written manually. Using functional testers requires huge capital investment over short periods of time since they become obsolete very quickly. Hence, Intel is now relying more on reusable low-cost testers.

As the data showed, manual test writing for future microprocessors is not feasible. Therefore, use of ATPG tools becomes essential to meet cost and time-to-quality requirements. Thus, the paradigm that has evolved is to use low cost structural testers and use ATPG to generate the required tests. The tests being generated are structural tests. The structural tests we generate differ from the classical structural tests in that we target defects via some novel fault models. We elaborate on this later in the paper. We next discuss the challenges that this paradigm shift brings with it.

Test Generation

The loss in accessibility to the I/O pins of the device has a major impact on the ability of engineers to write functional tests for the chip. It may be possible to

load functional tests through direct access to an on-chip cache, and run them from there, but it is difficult to generate tests that operate under this mode. As a result, most of the fault-grading tests that are applied through DFT methods have to be generated using ATPG tools.

ATPG for large high-performance designs poses unique problems. Today's microprocessors have multiple clock domains, operating at different speeds. Clock gating for low-power operation is pervasive. Typical designs have many complex embedded arrays that need to be modeled for the ATPG tool. Industry standard DFT techniques, such as full scan, are often too expensive in either die area, or performance or both [7].

Defect mechanisms in deep sub-micron designs are often manifested as speed failures under very specific conditions. Most commercial ATPG tools, which are based on the stuck-at and transition fault models, are not equipped to handle these complex failure modes.

Design for Test

Although ATPG technology has progressed during this time, the success of these tools is predicated on providing a high degree of access, controllability, and observability to the internals of the design by using DFT techniques.

Scan design, the best-known structured DFT technique, comes at the cost of both performance and area, although some trade-off is possible. In order to meet tight timing requirements, high-performance designs tend to have very few gates between storage elements, which results in a high latch-to-logic ratio. Therefore, implementing scan DFT generally translates into sacrificing considerable silicon real estate.

Another DFT technique that is gaining acceptance in the industry is Built-In Self-Test (BIST), which incorporates mechanisms to generate stimuli and compress responses for later off-chip comparisons into the design. BIST allows a large number of patterns to be applied at speed in a short time, with very little tester support. However, most logic BIST techniques that enjoy commercial success today require full scan, or close to it. In addition, they need design changes to enhance random-pattern testability, to allow at-speed test application, and to prevent the system from getting into an unknown state that can corrupt the compressed response.

Such intrusive DFT techniques cannot be applied across the board to high-performance devices, so logic BIST for microprocessors has only limited applicability today. High-volume, high-performance microprocessors have to choose between the high cost of scan DFT or resort to more custom access methods of getting stimuli to, and observing responses at, the boundaries of internal components.

Test Application Methodology

Industry data shows that testing a device using functional tests rather than other test patterns results in fewer escapes [4]. A possible explanation is that when the device is exercised in functional mode, defects that are not modeled, but affect device functionality, are screened out.

ATPG patterns differ fundamentally from functional test patterns: they explicitly target faults rather than checking for them by exercising the functionality of the device, and they are typically very efficient, detecting each fault fewer times in fewer ways. Also, since they are based on using DFT structures to apply tests, they are applied at a lower speed. Consequently, there is a risk of losing "collateral" coverage of defects that do not behave like the modeled faults.

Structural testers have a small set of pins that operate at a lower frequency than the device and contact only a subset of its I/O pins. The device needs to be equipped with special DFT access ports to load and unload the vectors from the tester. The boundary scan test access port, scan input and output pins, and direct access test buses are typically for this purpose.

A few seconds of functional test may apply millions of patterns to a chip. In contrast, due to power, noise, and tester bandwidth considerations, the serial loading of test vectors from the DFT ports may be slow, and the number of test vectors that can be applied from the structural tester may be far fewer than in a functional test environment. This has implications for the quality of the structural test set.

Speed Test

Unlike many standard parts, microprocessors are binned for speed. This necessitates speed test, where the objective is to determine the maximum frequency at which the part can be operated. In the past, a small set of the worst speed paths was identified, and tests written to exercise these paths were used to characterize the speed of the device. With increasing die sizes and shrinking device geometry, in-die process variation is becoming significant. It is no longer safe to assume that all paths will be affected equally, and a larger set of representative paths needs to be tested to determine the maximum operating frequency.

One of the implications of applying vectors in the DFT mode is that the device may not be tested in its native mode of operation. Special-purpose clocking mechanisms are implemented to apply the tests to the targeted logic blocks after they have been loaded. The electrical conditions, background noise, temperature, and power supply may all be different in the DFT mode.

These factors introduce inaccuracies, necessitating guard-bands, in measuring the speed of the device.

I/O Timing Test

Traditional I/O functional testing relies on the ability of the tester to control and observe the data, timing, and levels of each pin connected to a tester channel. The testing of the I/O buffers can be divided into three basic categories: timing tests (e.g., setup and valid timings), level tests (e.g., Vil and Vol specifications), and structural tests (e.g., opens and shorts). The timing specifications of the I/O buffers are tested during the class functional testing of the device. With the use of structural testers, dedicated pin electronics are no longer available on the tester to make timing measurements on each I/O pin on the device.

Assuming that the I/O circuit meets the design target and that timing failures are results of defects at the I/O circuits, the problem of testing complex timing becomes one of screening for these defects, instead of the actual timing specification itself.

Defect-Based Test

Applicability of the Stuck-At Fault Model

Although functional patterns are graded against the single stuck-at fault model, it is well known that most real defects do not behave like stuck-at faults. Instead, stuck-at fault coverage has been used as a stopping criterion for manual test writing with the knowledge that the functional tests would catch other types of defects that impact device functionality. This measure of test quality worked quite well for a long time. However, in the recent past, there is conclusive data from sub-micron devices that proves that the outgoing DPM can be further reduced by grading and developing functional tests using additional fault models such as bridges etc. Therefore, the success of the single stuck-at fault model cannot be guaranteed as we move further into the sub-micron devices.

The quality of ATPG patterns is only as good as the quality of the targeted fault models. As the test environment forces the transformation from functional to structural testing, there is yet another strong case for the development of better test metrologies than the simplified stuck-at fault model. Defect-based test addresses this risk by using better representations of the underlying defects, and by focusing the limited structural test budget on this realistic fault.

What is Defect-Based Test?

Before we define defect-based test, we distinguish between two terms: defect and fault model. Defects are physical defects that occur during manufacturing. Examples of defects are partial or spongy via, the presence of extra material between a signal line and the V_{dd} line, etc. Fault models define the properties of the tests that will detect the faulty behavior caused by defects. For example, stuck-at 1 tests for line *a* will detect the defect caused by a bridge between the signal line *a* and V_{dd}.

It has been reported in the literature [5] that tests that detect every stuck-at fault multiple times are better at closing DPM holes than are tests that detect each fault only once. This approach, called N-detection, works because each fault is generally targeted in several different ways, increasing the probability that the conditions necessary to activate a particular defect will exist when the observation path to the fault site opens up.

Defect-based tests are derived using a more systematic approach to the problem. First, the likely failure sites are enumerated. Each likely defect is then mapped to the appropriate fault model. The resulting defect-based fault list is targeted during ATPG. Tests generated in this way are used to complement vectors generated using the stuck-at fault model. Unlike the stuck-at model that works off of the schematic database, the starting point for defect-based test is the mask layout of the device under test. Layout-based fault enumeration is a cornerstone of defect-based test.

The use of better fault models is expected to enhance any test generation scheme (ATPG, built-in self-test, or weighted random pattern generation) because it provides a better metric for defect coverage than does the stuck-at fault model.

Although not a proven technology, defect-based test is a strong contender for addressing some of the risks of migrating from functional to structural test. The DBT effort at Intel is aimed at proving the effectiveness and viability of this approach. The following sections describe the key problems that have to be solved, the specific tooling challenges in automating defect-based test, and a system architecture showing DBT modules in the overall CAD flow.

Challenges of Defect-Based Test

Enumerating Defect Sites

The number of all possible defects on a chip is astronomical, and it is neither feasible nor worthwhile to generate tests for all of them. Fault enumeration is the task of identifying the most important defect sites and then mapping them into fault models that can be targeted by fault simulation and ATPG tools.

To enumerate likely defect sites, we need to understand the underlying causes of defects. Broadly speaking, defects are caused by process variations or random localized manufacturing imperfections, both of which are explained below:

- *Process variations* such as transistor channel length variation, transistor threshold voltage variation, metal interconnect thickness variation, and inter metal layer dielectric thickness variation have a big impact on device speed characteristics. In general, the effect of process variation shows up first in the most critical paths in the design, those with maximum and minimum delays.
- *Random imperfections* such as resistive bridging defects between metal lines, resistive opens on metal lines, improper via formations, shallow trench isolation defects, etc. are yet another source of defects. Based on the parameters of the defect and "neighboring parasitic," the defect may result in a static or an at-speed failure.

Techniques used for the extraction of faults due to random defects and process variations may differ, but the fundamental approach is to identify design marginalities that are likely to turn into defects when perturbed. The output of a fault extraction tool is typically ordered by probability of occurrence.

Defect Modeling

To test a device, we apply a set of input stimuli and measure the response of the circuit at an output pin. Manufacturing defects, whether random or systematic, eventually manifest themselves as incorrect values on output pins.

Fault simulators and ATPG tools operate at the logical level for efficiency. A fault model is a logic level representation of the defect that is inserted at the defect location. The challenge of fault modeling is to strike a balance between accuracy and simplicity as explained below:

- *Accuracy*. The output response of the logic-level netlist with the fault model inserted should closely approximate the output response of the defective circuit for all input stimuli.
- *Simplicity*. The fault model should be tractable, i.e., it should not impose a severe burden on fault simulation and ATPG tools.

During the model development phase, the effectiveness of alternative models is evaluated by circuit simulation. Vectors generated on the fault model are simulated at the circuit level in the neighborhood of the defect site, using an accurate device-level model of the defect. However, due to the number of possible defect sites and the complexity of circuit simulation, this can only be done for a small sample.

Defect-Based Fault Simulation

Simulation of defect-based models is conceptually similar to stuck-at fault simulation, with a couple of twists:

- The number of possible defect-based faults is orders of magnitude larger than stuck-at faults, so the performance of the tool is highly degraded. In order to be effective, a defect-based fault simulator has to be at least an order of magnitude faster.
- Defect-based faults may involve interactions between nodes across hierarchical boundaries, making it impractical to use a hierarchical or mixed-level approach to fault simulation. It is necessary to simulate the entire design at once, which also imposes capacity and performance requirements.

Defect-Based Test of Cache Memories

Background: The Growth of Caches for Microprocessors

The use of caches for mainstream microprocessors on Intel® architectures, beginning in the early 90s with the i486™ processor, heralded a return to Intel's original technical core competency, silicon memories, albeit with several new twists. The embedded CPU caches have increased in size from the 4K byte cache of the i486 processor generation to 10s and 100s of kilobytes on today's processors and to even larger embedded CPU caches being considered for the future. This has resulted in a steady increase in the fraction of overall memory transistors per CPU and in the amount of CPU cache die area throughout the last decade.

A second key cache test challenge is the increasing number of embedded arrays within a CPU. The number of embedded memory arrays per CPU has gone from a handful on the i486 and i860™ processors to dozens on the more recent Pentium® Pro and Pentium® II processor lines.

Memory Testing Fundamentals: Beyond the Stuck-At Model

The commodity stand-alone memory industry, i.e., DRAMs and 4T SRAMs, have evolved fairly complex sets of tests to thoroughly test simple designs (compared to the complexity of a modern microprocessor) [6]. The targeted fault behaviors include stuck-at, transition, coupling, and disturbs, and the resulting number of targeted tests per circuit, per transistor, or per fault primitive on a memory is much higher than for digital logic devices. On VLSI logic, the chal-

lenge is to achieve stuck-at fault coverage in the upper 90 percentile, while on stand-alone memories, the number of targeted tests per circuit component is typically in the 100s or more likely 1000s of accesses per bit within a robust memory test program.

One reason for the greater complexity of memory tests is that at the core of a typical digital memory is a sensitive, small signal bit, bit bar, and sense amp circuit system. Even for stand-alone memories, access and testing of the analog characteristics (e.g., gain, common mode rejection ratio, etc.) is not directly possible and must be done indirectly through the digital interface of address and data control and observability. A large number of first order variables subtly affect the observability of silicon memory defect behavior. Therefore, most memory vendors characterize each variant of a given product line empirically against a broad range of memory patterns before settling on the test suite that meets quality and cost considerations for high-volume manufacturing. These characterization test suites (also known as "kitchen sink" suites) consist of numerous algorithmic march patterns and different sets of cell stability tests (e.g., data retention, bump tests, etc).

A key concept for robust memory testing is the logical to physical mapping. On a given physical design of an array, the physical adjacencies and ordering of bits, bit lines, word lines, decoder bits, etc., typically do not match the logical ordering of bits (such as an address sequence from bit 0 to bit 1 to … highest order bit. Memory tests are designed to be specifically structural where worst-case interactions of the implemented silicon structures with true physical proximity are forced. Thus the true physical to logical mapping is a subsequent transform that must be applied to a given memory pattern in order to maximize its ability to sensitize and observe defects and circuit marginality. Correct and validated documentation to the downstream test writer of the actual physical-to-logical mapping is as important as other design collateral.

Embedded Cache Testing and DFT in the Context of Logic Technologies

Testing of embedded caches also needs to consider the context of related logic technologies. To start with, the basic embedded cache memory cell is typically a six transistor (6T) SRAM as compared to the more typical DRAMs and four transistor (4T) SRAM of the stand-alone silicon memory industry. The 6T SRAM offers better robustness against soft errors and can be thoroughly tested to acceptable quality levels with somewhat simpler test suites. However, the critical motivating factor is that a 6T SRAM cell is feasible, within the context of a high-performance logic silicon fabrication process technology, without additional process steps.

The smaller size (area, # bits) and 6T cell of the embedded CPU cache make it less sensitive than the stand-alone commodity 4T SRAMs and DRAMs. This is somewhat offset by the fact that embedded caches are generally pushing the SRAM design window on a given fabrication technology for system performance reasons. Therefore, adequate testing of embedded 6T SRAMs requires an optimal use of robust memory test techniques targeted at defect behaviors, such as complex march algorithms and cell stability tests.

A critical challenge for embedded SRAM caches is the architectural complexity of access and observability of such arrays compared to a stand-alone memory. For example, for an embedded array such as an instruction cache or a translation buffer, there may not be a normal functional datapath from the array output to the chip primary outputs, making writing of even the simplest memory algorithmic patterns such as the 10N March C- an extreme challenge for even the most experienced CPU design engineers and architects.

In the end, the number and variety of caches and embedded arrays in today's microprocessors demand a multiple of DFT and test solutions optimized to the physical size and area of the various arrays, the performance and cost boundary conditions, and the architectural and micro-architectural details of an embedded array's surroundings. Circuit-level DFT, such as WWTM [7], can offer targeted structural coverage, in this case against cell stability issues and weak bits. External access via special test modes or self-test (BIST) circuits may provide the better solution within different sets of variables. However, care must be taken to ensure the completeness and correctness of the solution in any case and that some level of structural approach is used, i.e., appropriate stimulus-response mapped to the physical implementation of the memory structures. Different types of memory structures, e.g., small signal SRAMs, full Vcc rail swing CMOS register files, CAMs, or domino arrays, each require a targeted structural approach mapped to their strength and weaknesses with respect to defect response.

Technology Development Strategy

The technology for defect-based test spans multiple disciplines in design, CAD tooling, and manufacturing. Although individual components have been tried both within Intel as well in academia and industry, real data on high-volume, high-performance microprocessors is needed to establish the value of this approach.

The defect-based test program at Intel emphasizes early data collection on the effectiveness of fault models. Partnerships with design teams interested in pioneering these new capabilities as they are developed

form a cornerstone of this effort. Technology development proceeds in phases as follows:

- *Fault model development*. There are a large number of possible defect types that can be modeled. Defects are chosen for modeling based on frequency of occurrence, ease of modeling, escape rate, and perceived importance to the partner design team. Bridges and path delay faults will be the first set of fault models to be investigated.
- *Tool development*. A minimal set of prototype tools is developed for the enumeration and simulation of the target fault models. These tools are targeted for limited deployment to a select group of experts in the project design team. The focus of tool development is on accuracy, not performance. Where possible, the tools are validated against existing "golden" capabilities.

  Tools for defect enumeration need to leverage physical design and performance verification tools. Close co-operation with tool builders and project design automation teams is required to build on existing tools, flow, and data in order to facilitate the defect-extraction process.
- *Enumerating fault sites*. The actual task of enumerating fault sites is performed jointly by the technology development team and the design team. Working together, test holes such as new architectural enhancements or modules for which legacy tests could not be effectively ported are identified. Fault grading resources are allocated for defect-based test on those regions. When available, data from the FABs are used to assign probabilities to defect sizes.
- *Test generation*. Defect-based tests are generated by first grading functional validation and traditional fault grade vectors, and then by targeting the undetected faults for manual test writing. Test writing is necessary at this time because a defect-based ATPG is not yet available, and the legacy designs on which the technology is being pioneered do not have adequate levels of structured DFT. To contain the cost of test writing, defect-based tests are written for carefully selected modules of the design.
- *Model validation.* Model validation requires close partnering with the product engineering team. Some changes are required to the manufacturing flow to collect data on the unique DPM contribution of the defect-based tests.

  Data from the model validation phase is fed back into model development, as illustrated below. Once a particular fault model is validated, we will enter into development (or co-development with a tools' vendor) of an ATPG capability for that model.

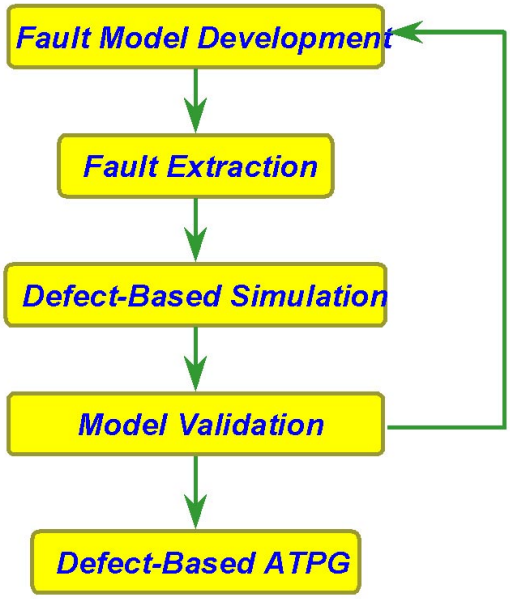

Figure 4: Technology development process flow

Defect Modeling

Modeling of Random Defects

The challenge in fault modeling is to capture a general cause and effect relationship that can be easily simulated or targeted in the case of automatic test pattern generation. A degenerate case of this general approach is a line stuck-at fault model where output at a node is always a logical zero or always a logical one regardless of the logic value it is driven by. Another popular fault model that has been used to target random speed failures is the transition fault model, which is essentially a stuck-at fault with the addition of the condition that the faulty node make a transition, i.e., be at the opposite logic value in the cycle prior to detection.

In creating a realistic fault model for a defect, we must avoid explicitly tabulating the behavior of the defect for every state of the circuit. A table-driven approach will not lend itself to a scalable automated solution for design sizes that exceed 5 million primitives. The approach we use here is to transcribe the deviation in analog behavior into simple conditional logical deviations.

There are a large number of possible failure mechanisms that cause random defects. Rather than de-

velop models for them all and then launch into model validation, our approach is to stage the development of models and tools to address defect types in the order of their importance, and to intercept designs with a complete prototype flow for each model as they become available. This allows us to collect data on the DPM impact of defect-based test early on, and it provides feedback that we can use to refine our models.

One of the most common defect types today is interconnect bridges. As metal densities increase, the importance of metal bridges as a defect-inducing mechanism will grow. Interconnect bridging defects exhibit a range of behavior based on different values of bridge resistance.

This effect is illustrated for the circuit in Figure 5. There is a bridge defect between node *j* and *k* in this example. Node *k* is held at logic 0 as *j* changes from 0 to 1. The signal transition is propagated and observed at output *v*.

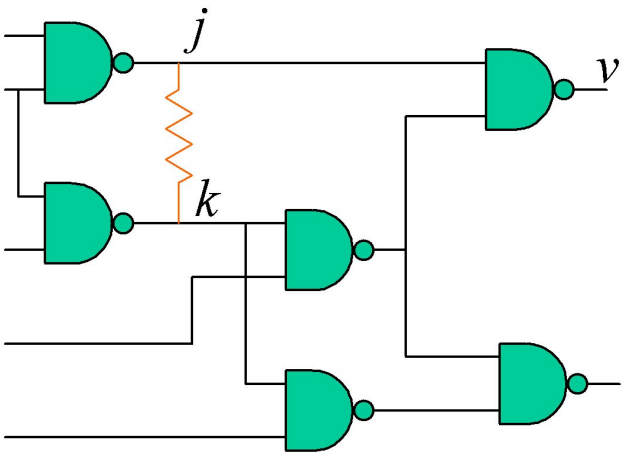

Figure 5: Example circuit with a bridge defect

Figure 6 shows the output response of the circuit for different values of the bridge resistance. Threshold voltages are marked using horizontal dashed lines, and the vertical dashed line shows the required arrival time at node v for the transition to be captured in a downstream latch.

The plot shows three distinct circuit behaviors for varying bridge resistance. For low resistance values, the output never reaches the correct logic value, and the defect shows up as a static logic failure. For intermediate resistances, the output goes to logic 0 too late, resulting in a speed failure. Very high bridge resistances are benign from the viewpoint of correct logical operation of the circuit.

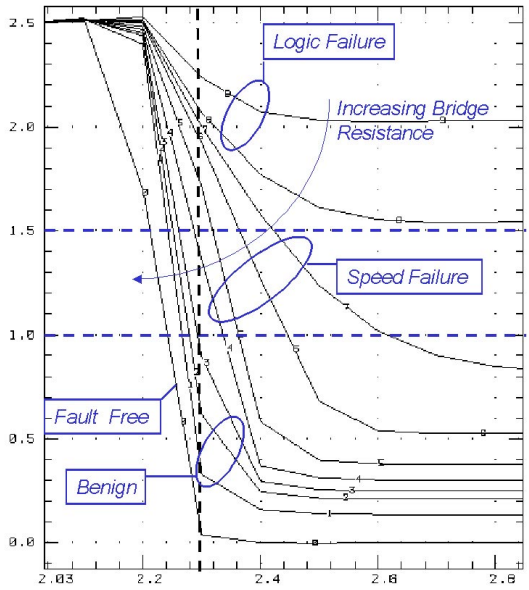

Figure 6: Output responses for a range of bridge resistance

For low resistances, the defect can be modeled as node *j* stuck at logic 0 with the condition that *k* is at logic 0. Speed failures can be modeled as a slow-to-rise transition at node *j*, with the condition that *k* is held at logic 0. Such fault models, based on generalizations of the conventional stuck-at and transition faults, are called constrained fault models.

As feature sizes are scaled down, the metal pitch is reduced in tandem to increase density. Reduced metal pitch in turn imposes limitations on the height of metal interconnects that must also decrease to improve manufacturability. Thus the line resistance per unit length goes up almost quadratically. Sustained yield requirement dictates that defect densities remain the same, which in turn implies that interconnect bridge defects also scale in dimension. Higher bridge resistance coupled with lower device resistance during ON state results in more speed failures than hard failures as illustrated in Figure 6 above.

Modeling of Systematic Defects

Not all defects are of a random nature. Known factors such as reticule position, die location on a wafer, mask imperfections, polysilicon density, device orientation, etc., cause systematic variation across wafers and dice. These effects are expected to gain prominence due to reduced noise tolerance as well as a general increase in systematic variability because of such factors as migration towards 300mm wafers, lithographic equipment, and material re-use.

Steeper production ramps are putting increasing pressure on cutting down the time for design correction and test creation based on silicon data. Thus modeling such effects is critical to the success of test.

Process variations lead to delay problems. Therefore, using information of process variations in speed test target selection needs to be addressed.

Defect-Based Test Tooling Challenges

Defect Enumeration

The goal of defect enumeration is to prune the list of all possible defects to a manageable number of the most likely faults. Because the likelihood of a fault has a strong dependence on layout geometry, process parameters and timing marginality, defect enumeration is a multi-disciplinary problem.

Here we describe layout-driven and timing-driven approaches to fault enumeration, and we discuss the inherent challenges.

Physical Design Inductive Fault Analysis

Inductive Fault Analysis (IFA) is based on the premise that the probability of a defect occurring at a particular site is a function of the local layout geometry and the distribution of failure mechanisms observed for the manufacturing process. The most commonly observed defects can be classified into two broad categories of physical faults:

- Bridges occur when the defect causes a conducting path between two nodes that are electrically isolated by design. The resistance of the bridge can vary by process, layer, and defect mechanism.
- Breaks happen when the defect introduces undesired impedance along a conducting path. In an extreme case, a break can result in an open circuit.

These physical fault models are then mapped onto logical fault models that can be used for fault simulation at the logical, or gate level, of abstraction. If the likelihood of the defect mechanism causing opens and breaks is known for the process, the physical fault sites extracted by IFA are weighted by probability. These probabilities can be used for pruning the fault list, and for expressing the fault coverage obtained by fault simulation in terms of the overall probability of catching a defective part. This weighted fault coverage number can be a better predictor for outgoing DPM than stuck-at fault coverage.

Traditionally, IFA has focussed on layout geometry and defect distribution, and it has ignored the testability of a fault. This last parameter is an important one: If the faults identified using IFA are highly testable, i.e., easily covered by tests for stuck-at faults, then using an IFA-based approach will not yield a significant incremental DPM improvement over a standard stuck-at fault model. Examples of highly likely and highly testable faults are bridges to power rails and clock lines. Therefore, the challenge for effective IFA tools is to identify faults that are both highly likely and relatively difficult to detect using stuck-at fault vectors.

Because they work at such a low level of abstraction, IFA tools need to be scalable in order to be effective on increasingly larger designs. Two divide-and-conquer approaches can be applied to the problem:

- *Hierarchical analysis*. This is where layout blocks are analyzed at a detailed level for bridges and breaks on cell-level nodes, and at a global level to analyze inter-block connectivity. The obvious drawbacks of this method are that interactions between wires across blocks, and between block-level and chip-level layout, are ignored. This problem is accentuated by the increasing trend toward over-the-cell global routing.
- *Layout carving, or "cookie-cutting."* In this approach, the layout is flattened and carved into manageable pieces called "cookies." Each cookie includes the layout to be analyzed, as well as sufficient surrounding context. A second phase is required to roll up the results collected at the cookie level, and to tie up the inter-cookie interactions.

Timing-Driven Analysis

As mentioned in a previous section, the performance verification tools for large microprocessor designs are not entirely fool proof. To begin with, the PV database is made up of data from different sources, some of which are SPICE-like simulations (very accurate) and some of which are simple estimators. The net result of this could be incorrectly ordered critical paths (speed-limiting circuit paths). During silicon debug and characterization, some of these issues are generally uncovered.

However, some serious issues abound as we look into the future. First, the increased on-die variation in deep sub-micron technologies means that different paths on the chip can be impacted differently. Further, the trend towards higher frequencies implies fewer gates between sequential elements, which may lead to a larger proportion of the chip's paths having small margins. These two factors combined pose one of the biggest test challenges, namely, speed test.

It is no longer just sufficient to have a few *most critical paths* in the circuit characterized during silicon debug. What is required is an automatic way to enumerate all such paths and then grade the structural tests for "path delay fault" coverage. There are two main issues that need to be solved. First, PV tool limita-

tions need to be worked around (issues related to generating an ordered list of critical paths), and second, modeling issues related to mapping of paths from transistor level to gate level need to be resolved. (Fault simulation happens at the gate level.)

It is likely that this huge path list can be pruned to a more manageable size. Paths could be selected based on their criticality of speed to the design and on their diversity in composition in terms of distribution of delay amongst various constituent factors such as delays on all interconnect layers and actual devices.

Comprehensive Defect Enumeration

While layout analysis may identify potential bridge defect sites, a resistive bridge may not always manifest itself as a logic error. An example of such a situation would be if the defect site has adequate slack designed into it, an increase in delay up to the slack amount will not be ordinarily detectable. Slack may change with a change in cycle time or a change in power supply voltage, thus altering the test realities.

It is therefore required that the defect enumeration scheme be coupled with timing analysis tools, which in turn should be designed to understand the effect of the test environment (temperature, voltage, cycle time) on slack.

Defect-Based Simulation and ATPG

Traditional test automation tools need to be rethought in the context of defect-based test. The fundamental reason for the effectiveness of the stuck-at fault model is that it opens up an observation path starting from the fault site. Unfortunately, the conditions needed to cause the erroneous circuit behavior may not be created at the time the observation path is set up.

Data reported in the literature show that the effectiveness of a test set could be improved by including vectors that detect the same stuck-at fault multiple times, in different ways. This approach, called N-detection, is a random way to set up the conditions needed to activate different failure modes. Defect-based fault models take this notion a step further by specifying the actual excitation conditions, called *constraints*.

- *Excitation conditions.* These are a relatively straightforward extension to commonly used fault models. Constrained stuck-at and constrained transition faults behave like their traditional counterparts except that the fault effect becomes manifest only when an externally specified condition is met.

  Existing fault simulation and test-generation tools can be used to simulate these models by augmenting the target netlist to detect the excitation condition and to inject the fault when it occurs. However, this can be expensive in terms of netlist size for big designs. Also, depending on the location of the set of nodes involved in the constraints and the fault location, the augmenting circuitry can cause design-rule violations such as phase coloring.

- *Propagation conditions*. Certain types of physical faults (such as highly resistive bridges and opens) can manifest themselves as localized delay defects. However, the size of the delay is not always large enough to allow it to be treated as a transition, or gross delay. In such cases, the effectiveness of the test can be increased, propagating the fault effect along the paths with the lowest slack. This method implies a tie-in to the timing analysis sub-system.

- *Path delay fault simulation*. Several path delay fault models have been proposed in the literature with a view to identifying tests that are robust (less susceptible to off-path circuit delays), and to simplifying the model to ease fault simulation and test generation. Any of these fault models can be used, but there are two new considerations:

  Paths in high-performance designs are not always limited to a single combinational logic block between two sequential elements. A path can span multiple clock phases, crossing sequential elements when they are transparent. A practical path delay fault model should therefore be applicable to multicycle paths. Note that such paths may feed back onto themselves (either to the source of the path or to an off-path input).

  The second consideration is that fault simulation and ATPG are typically performed at the gate level, whereas paths are described at the switch level. When a switch-level path is mapped to the gate level, a path may become incompletely specified. There may be multiple ways to test the same gate-level path not all of which exercise the same switch-level path. This problem can be addressed by specifying gate-level conditions that will exercise the switch-level path in a manner analogous to specifying excitation conditions for random defects.

- *Circuit design styles*. High-performance designs have core engines running at very high speeds and external interfaces running at lower speeds. In addition, there may be internal subsystems that run at a different clock frequency. Test generation and fault simulation tools have to be designed to accommodate multiple clock domains running at different frequencies. The clocks are typically generated internally and synchronized. DFT design rules, particularly those that check the clocking methodology, need to be enhanced to handle such designs.

Another important design consideration is power delivery and consumption. In order to reduce a chip's power needs, clocks are often gated to dynamically turn off units that are not being used at a particular time. In the past, many tool designers assumed that clock-gating logic could be controlled directly by external pins, or they treated clock-gating logic as untestable. These assumptions are no longer valid.

- *Capacity and performance*. Next-generation CPUs are expected to require 5 to 10 million primitives to model at the gate level. The designs contain on the order of a hundred embedded memory arrays. These arrays have multiple read/write ports, with some ports accessing only parts of the address or data spaces of the array. In the past, most ATPG tools have provided support for simple RAM/ROM primitives that can be combined to model more complex arrays. However, from the point of view of database size and test generation complexity, it is essential to directly support more general behavioral models.

  Defect-based fault models impose additional performance requirements on the tools because of the exploding number of faults that need to be targeted. In order to deal with larger designs, shrinking time-to-quality goals, and the larger number of faults, the performance of test automation tools needs to increase by an order of magnitude.

Failure Diagnosis

Automated failure diagnosis is valuable at different stages of a product's life: silicon debug and qualification manufacturing test and analysis of customer returns. Next-generation failure analysis tools have two major requirements:

- They must support defect-based models. Diagnostic tools need to leverage the defect resolution provided by the new fault models. This will enhance diagnostic resolutions by narrowing down the probable cause of a failing device to one defect-based fault, where partial matches were found, before using the stuck-at fault model. Diagnostic resolution can be further enhanced by the use of defect probability for prioritizing candidate failures.
- They must support limited sequentiality for high-performance designs that cannot afford scan DFT in pipelined stages.

Defect-Based Tooling Framework

The design flow in Figure 7 shows the new CAD modules introduced for DBT and their relationship to existing design and test automation modules. The new modules are highlighted in yellow.

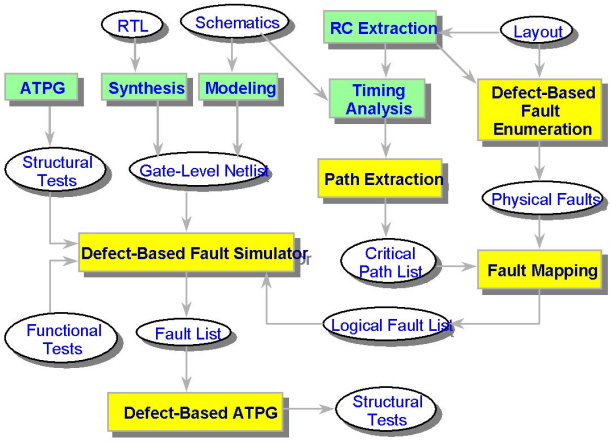

Figure 7: Defect-based test system architecture

The left half of the flow is analogous to the traditional fault simulation and ATPG flow. These tools work on a gate-level model, which is generated either top-down by synthesis of RTL, or bottom-up by logic modeling of device-level circuits. The defect-based fault simulator accepts fault lists of realistic defect models. Traditional ATPG vectors, as well as existing functional tests, are fault simulated to filter out defect-based faults that are detected by these tests. A defect-based ATPG is used to generate tests for undetected faults.

The right half of the flow is for layout and timing-driven fault enumeration, and it is new to DBT. The analogous step for traditional ATPG is stuck-at fault enumeration and collapsing based purely on gate-level analysis. Random faults are typically enumerated from the layout with the possible use of interconnect capacitances obtained by RC extraction tools. Critical paths for speed test are extracted from timing analysis. The identified fault sites exist at the layout or device level, and they need to be mapped to the logical level for fault simulation and ATPG.

Conclusion

In this paper we described the challenges faced by Intel in continuing with functional test as the primary mechanism for screening manufacturing defects, and we examined structural test as an alternative. Three major test quality risks were identified in migrating to structural test:

- Reduced test data volume due to the inefficiencies in loading test patterns from a structural tester.

- The loss of collateral defect coverage provided by functional tests that are applied at speed in the normal functional mode of operation.
- Sub-micron trends indicating that interconnect defects such as bridges and opens will dominate the defect distribution. Simulation results were presented that indicate that an increasing number of defects will result in speed failures rather than hard failures, requiring alternate ways of generating test patterns.

Defect-based test was introduced as an approach to mitigate some of these risks by increasing the effectiveness of ATPG-generated vectors. While this approach is intuitively appealing, it poses formidable challenges. Little hard evidence is available on the effectiveness of such an approach. Fault models that represent defect behavior well, and are tractable from a test generation viewpoint, have to be developed. Tools for the enumeration of likely fault sites and for test generation tools with the new fault models need to be implemented.

The technology development strategy for DBT was presented as an evolutionary cycle that builds on prototype capabilities and uses strategic partnerships with design teams. Silicon data collected from these experiments are used to refine and validate fault models and the tooling collateral as they are developed.

The tooling challenges for defect-based test for large, high-performance designs were discussed. Commercial capabilities that exist today are either insufficient, or cannot be scaled to meet the needs of next-generation microprocessor designs. These challenges span the design flow from logical to physical design, and they will require a concerted effort by the CAD industry to make defect-based test a robust, scalable solution.

Acknowledgments

The defect-based test effort spans three departments in MPG and TMG. We thank Will Howell and Paul Ryan of TT Q&R, Sujit Zachariah, Carl Roth, Chandra Tirumurti, Kailas Maneparambil, Rich McLaughlin, and Puneet Singh of Test Technology, and Mike Tripp, Cheryl Prunty, and Ann Meixner of STTD for their ongoing contributions.

We have benefited greatly from feedback received over the course of several technology reviews held with the Willamette DFT team, in particular Adrian Carbine, Derek Feltham, and Praveen Vishakantaiah.

References

[1] Schnarch, Baruch, "PP/MT Scoreboarding: Turning the Low DPM Myth to Facts," *Proceedings of the 1998 Intel Design and Test Technology Conference*, pp. 13-18, Portland, OR, July 21-24, 1998.

[2] Wayne Needham, internal memo based on data extracted from 1997 SIA Roadmap.

[3] Design and Test Chapter, *National Technology Road Map*, 1997, available on the Web (URL: www.sematech.org).

[4] "Sematech Test Method Evaluation Data Summary Report, " Sematech Project S-121, version 1.4.1, 1/30/96.

[5] S.C. Ma, P. Franco, and E.J. McCluskey, "An Experimental Chip to Evaluate Test Techniques Experiment Results," *Proceedings 1995 Int. Test Conference*, pp. 663-672, Washington, D.C., Oct. 23-25, 1995.

[6] A. J. van deGoor, *Testing Semiconductor Memories, Theory and Practice,* John Wiley and Sons, Ltd, England, 1991.

[7] A. Meixner and J. Banik, "Weak Write Test Mode: An SRAM Cell Stability Design for Test Technique," *Proc. 1996 Int. Test Conf.*, pp. 309-318.

[8] A. Carbine and D. Feltham, "Pentium® Pro Processor Design for Test and Debug," *IEEE International Test Conference*, 1997, pp. 294-303.

Authors' Biographies

Sanjay Sengupta received an MSEE in electrical and computer engineering from the University of Iowa and a B.E. (Hons.) in electrical and electronics engineering from BITS, Pilani, India. He works on the development of test tools and methodology for next-generation microprocessors, and currently manages the defect-based logic test effort. Prior to joining Intel, he worked on test automation at Sunrise Test Systems and LSI Logic. His e-mail is sanjay.sengupta@intel.com.

Sandip Kundu has a B.Tech (Hons.) degree in electronics and electrical communication engineering from IIT, Kharagpur and a PhD in computer engineering from the University of Iowa. Prior to joining Intel, he worked at IBM T. J. Watson Research Center and at IBM Austin Research Laboratory. Sandip has published over forty technical papers and has participated in program committees of several CAD conferences including DAC, ICCAD, and ICCD.
His e-mail is sandip.kundu@intel.com.

Sreejit Chakravarty received his BE in electrical and electronics engineering from BITS, Pilani and a PhD in computer science from the State University of New York. He joined Intel in 1997. Prior to that, from 1986-97, he was an Associate Professor of Computer Science at the State University of New York at Buffalo. Sreejit has published over 70 technical papers and has served on the program committee of several international conferences, including VTS and VLSI Design. He has co-authored a book on I testing. His e-mail is sreejit.chakravarty@intel.com.PPQ

Praveen Parvathala received an MSEE from New Jersey Institute of Technology in 1989. Since then he has been working at Intel. As a designer, he contributed to the development of the i860™, Pentium® and Merced™ microprocessors. Since 1995 he has been working on DFT and is currently involved in the development of defect-based DFT methods and tools in MPG's Test Technology group.
His e-mail is praveen.k.parvathala@intel.com.

Rajesh Galivanche received his MSEE from the University of Iowa in 1986. Since then he has worked in the areas of Design for Test, ATPG, and simulation. Previously, Rajesh worked at Motorola, LSI logic, and Sunrise Test Systems. Currently, he manages the Logic Test Technology development team in the Microprocessor Product Group.
His e-mail is rajesh.galivanche@intel.com.

George Kosonocky received his BSEE from Rutgers University. He presently manages MPG's Test Technology organization and has held various management positions at Intel since 1983 in design engineering, marketing, quality and reliability, and program management. George holds a patent in non-volatile memory design. His e-mail is george.a.kosonocky@intel.com.

Mike Rodgers received a BSEE from the University of Illinois in 1986. He has been with Intel for 12 years in various capacities in Microprocessor Q&R including managing A4/T11 Q&R and FA groups in Santa Clara and IJKK. He currently is responsible for TT's Cache Technology and co-chairs the TMG-MPG Test Technology Planning Committee.
His e-mail is michael.j.rodgers@intel.com.

TM Mak is working on circuit and layout-related DFT projects. He has worked in product engineering, design automation, mobile PC deployment, and design methodology for various products and groups since 1984. He graduated from Hong Kong Polytechnic in 1979. His e-mail is t.m.mak@intel.com.