AN IMPROVED METHOD AND APPARATUS FOR AUTOMATED DESIGN AND VERIFICATION OF INTEGRATED CIRCUITS

Dragomir M. Nikolic
University of Kentucky, Dgn@cypress.com

Recommended Citation
https://uknowledge.uky.edu/gradschool_theses/268
AN IMPROVED METHOD AND APPARATUS FOR AUTOMATED DESIGN AND VERIFICATON OF INTEGRATED CIRCUITS

by

Dragomir M. Nikolic

This thesis is submitted to the faculty at the College of Engineering, University of Kentucky in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

Director: Dr. J. Robert Heath, Associate Professor of Electrical and Computer Engineering

Lexington, Kentucky
2005

Copyright © Dragomir Nikolic 2005
Electronic Design Automation (EDA) tools have always played an important role in Very Large Scale Integrated (VLSI) Circuit development. Two major linked roles are to reduce total design cycle time of Integrated Circuits (ICs) and increase profitability. Profitability can significantly be increased through quicker development and shorter time to market which in turn can be achieved through design automation. In addition to design cycle time reduction, design automation provides consistency and repeatability, which are critical for capturing and sharing of efficient design methods. Application Specific Integrated Circuit (ASIC) development and verification is a very manual and time consuming process. The objective of this research is to improve a design’s parameters, such as silicon area, timing and power, while achieving design cycle time reduction, through design and implementation of a new action and event-driven flow automation tool. This novel approach to providing a quick and easy method for rapid prototyping and automated verification without need for upgrading individual tools, can lead to great design improvements with very limited upfront investment. The automation tool is developed using a range of programming languages including SKILL (a derivative of LISP), C/C++ and Tool Command Language (TCL). Design and implementation, via use of the tool, are verified through hardening of a 32 bit, low power processor core block while showing design cycle time speedup >7x and reduction in total chip area by more than 20%. Supported and used software tools include: Cadence Silicon Ensemble Place and Route, Signal Integrity, Physical Verification, Cross-Talk Analysis and Abstract Generation; Synopsys Static Timing Analysis (STA); Mentor Graphics Physical Verification, Parasitic Resistance and Capacitance (RC) Extraction.
First of all, I want to thank my wife Heather and our three boys, Vladislav, James and Gabriel for their support and patience during my graduate studies and this research. Secondly, I would like to thank Cypress Semiconductor for the opportunity to participate in the described cutting edge research and development, numerous reviews with members of the design community as well as their provision of numerous computing and software resources. Thirdly I would like to thank Ronald Kalim, my co-worker and friend who participated on this project and helped develop code implementation for my software design. I would also like to thank my advisor Dr. J. Robert Heath for his input, guidance and support during this project. Finally, I would like to thank members of my committee Dr. Henry Dietz and Dr. Joseph Elias for their time, feedback and guidance.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................ III
TABLE OF CONTENTS ......................................................................................................... IV
TABLE OF FIGURES ........................................................................................................ VII

CHAPTER 1. INTRODUCTION ............................................................................................... 1
  1.1 Background ............................................................................................................. 1
  1.2 Problem Statement ............................................................................................... 2
  1.3 Additional ASIC Flow Background ..................................................................... 4
    1.3.1 Past ASIC Design Cycle ............................................................................ 4
    1.3.2 Present ASIC Design Cycle ....................................................................... 5
    1.3.3 Future ASIC Design Cycle ......................................................................... 7
  1.4 Definition of Terms .............................................................................................. 7

CHAPTER 2. REQUIREMENTS ........................................................................................ 11
  2.1 Scope .................................................................................................................. 11
  2.2 Customers ........................................................................................................... 11
  2.3 User Requirements ............................................................................................. 11
    2.3.1 Functional Requirements ......................................................................... 11
    2.3.2 Environment and Hardware ..................................................................... 16
    2.3.3 Performance and Capacity ...................................................................... 17

CHAPTER 3. ARCHITECTURE ....................................................................................... 18
  3.1 Knowledge Bases ............................................................................................... 19
    3.1.1 Vendor Tool Knowledge Base .................................................................. 19
    3.1.2 Design Knowledge Base ........................................................................... 19
    3.1.3 Technology Knowledge Base ................................................................... 20
    3.1.4 Tool Knowledge Base .............................................................................. 20
    3.1.5 Project Knowledge Base .......................................................................... 20
    3.1.6 User Knowledge Base ............................................................................. 20
  3.2 Data Flow ............................................................................................................. 20
    3.2.1 Endcap Cell Insertion .............................................................................. 25
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASIC Design Cycle</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>WLM vs Physical Synthesis</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Current ASIC Design Cycle</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>MakeBlock Architecture</td>
<td>18</td>
</tr>
<tr>
<td>5a</td>
<td>MakeBlock Default Data Flow</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>Full Floorplan</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Endcap Cells</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Paths not Covered by SDF</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>Filler Cells, Power Rails and Routed Signals</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>Antenna and Layer Hopping Diagram</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>SI Functional Failure [10]</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>SI Effect on Signal Delay [10]</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>MakeBlock Top-Level Log File</td>
<td>45</td>
</tr>
<tr>
<td>14</td>
<td>Siggen Module Default Log File</td>
<td>46</td>
</tr>
<tr>
<td>15</td>
<td>Abstract Generator Compliance Checker</td>
<td>50</td>
</tr>
<tr>
<td>17</td>
<td>Connectivity Addition Form</td>
<td>52</td>
</tr>
<tr>
<td>18</td>
<td>DEFIn Repair Form</td>
<td>54</td>
</tr>
<tr>
<td>19</td>
<td>Met1-Met2 Contact Pcell</td>
<td>55</td>
</tr>
<tr>
<td>20</td>
<td>Met1-Met2 Contact P-cell Properties Form</td>
<td>56</td>
</tr>
<tr>
<td>21</td>
<td>Power/Ground Bus Connections</td>
<td>58</td>
</tr>
<tr>
<td>22</td>
<td>Non-Default Rule Via Contacts</td>
<td>60</td>
</tr>
<tr>
<td>23</td>
<td>Default Via Contact</td>
<td>61</td>
</tr>
<tr>
<td>24</td>
<td>ARM7 Organization and Architecture [23]</td>
<td>66</td>
</tr>
<tr>
<td>25</td>
<td>ARM7 Instruction Set Quick Reference Guide [23]</td>
<td>67</td>
</tr>
<tr>
<td>26</td>
<td>nnARM Core Floorplan Initialization Form</td>
<td>69</td>
</tr>
<tr>
<td>27</td>
<td>nnARM Core Floorplan</td>
<td>70</td>
</tr>
<tr>
<td>28</td>
<td>nnARM Core MakeBlock Top-Level Log File</td>
<td>81</td>
</tr>
<tr>
<td>29</td>
<td>nnARM Core Ctgem Report</td>
<td>82</td>
</tr>
<tr>
<td>30</td>
<td>nnARM Core Detailed Clock Timing Report</td>
<td>83</td>
</tr>
<tr>
<td>31</td>
<td>External Wrapper for Launching MakeBlock Runs</td>
<td>84</td>
</tr>
<tr>
<td>32</td>
<td>Design Optimization Results – 70% Utilization</td>
<td>85</td>
</tr>
<tr>
<td>33</td>
<td>Design Optimization Results – 80% Utilization</td>
<td>85</td>
</tr>
<tr>
<td>34</td>
<td>Design Optimization Results - 85% Utilization</td>
<td>86</td>
</tr>
<tr>
<td>35</td>
<td>Design Optimization Results - 91% Utilization</td>
<td>86</td>
</tr>
<tr>
<td>36</td>
<td>Design Optimization Results - 92% Utilization</td>
<td>87</td>
</tr>
<tr>
<td>37</td>
<td>Design Optimization Results - 93% Utilization</td>
<td>87</td>
</tr>
<tr>
<td>38</td>
<td>Design Optimization Results - 94% Utilization</td>
<td>88</td>
</tr>
<tr>
<td>39</td>
<td>Design Optimization Results – 95% Utilization</td>
<td>88</td>
</tr>
<tr>
<td>40</td>
<td>Physical Placement Summary at 94% Utilization</td>
<td>89</td>
</tr>
<tr>
<td>41</td>
<td>Clock Tree Insertion Failure Summary</td>
<td>89</td>
</tr>
<tr>
<td>42</td>
<td>Failed Physical Placement Summary at 95% Utilization</td>
<td>89</td>
</tr>
<tr>
<td>43</td>
<td>Failed Physical Placement Detailed Log at 95% Utilization</td>
<td>90</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

1.1 Background

The Very Large Scale Integrated (VLSI) Circuit industry for many years has been using automation and Place-and-Route tools for development and verification of Application Specific Integrated Circuits (ASICs). This process is commonly referred to as RTL (Register Transfer Level) to GDS (Graphic Design Station file format) flow, which is used to take a design described in Hardware Description Language (HDL), such as Verilog or VHDL, and turn it into functional silicon. As design complexity increased as did the need to shorten design cycle time and achieve first pass functional silicon, “RTL to GDS” flow went through various changes and modifications [1, 2].

The high level basic flow consists of HDL development and its synthesis into gate level netlist, followed by floorplan design and cell placement, clock tree insertion, routing, signal integrity correction and finally timing and physical verification as shown in Figure 1.
Several problems with this design methodology can be easily identified. Firstly, each of the steps of the design process is very manual and requires significant amount of user time and user expertise in order to be accomplished correctly. Secondly, flow is very much trial-and-error based and provides very little information about problems, which could be encountered in the later design steps.

1.2 Problem Statement

In recent years, with ever increasing design complexity, shrinking technologies,[1, 2], need for shorter design cycle time and faster time to market, the challenge is greater than it has ever been to Tape-Out (TO) first-pass silicon in the shortest amount of time possible. However, following Moore’s Law [1], the exponential rise of design complexity [1, 2] coupled with newly encountered challenges in a Deep Sub-Micron (DSM)
design, such as cross-talk signal integrity [3], proved that existing flows are no longer sufficient.

The Electronic Design Automation (EDA) industry has been struggling to keep up with challenges and has been typically slow to respond with integrated, quality solutions to newly encountered problems in DSM such as Signal Integrity and Timing Closure. For this reason many chip manufacturers have been forced to dedicate a large share of their R&D budget to invent and implement new design automation tools and methodologies.

Tools and techniques used for development and hardening of integrated circuits with large amounts of digital logic are very manual in nature and demand a high level of expertise and experience from users before they can be used with efficiency and effectiveness. Given VLSI Circuit industry trends and the need for continuous design and time to market reduction, flexible automation tools must be developed which will streamline development and leverage strengths of point tools (tools performing individual steps of the flow) throughout the design cycle. The tool to be developed as part of this research, named MakeBlock, will accept design and timing constraints as input, and output physically clean GDS representation of the design while meeting timing requirements.

MakeBlock will provide a framework and an automated execution environment with a goal of enabling rapid design prototyping. Rather than focusing on a specific point tool and point algorithm in order to achieve a most efficient and most optimized design, MakeBlock will focus on ease of use, design automation and ability to complete many “what-if” spins through the design cycle thus providing a most optimal achievable solution. In addition, the MakeBlock automated execution environment is characterized by two unique concepts, action driven flow and event driven flow.
Most tools force users into a predictable flow of steps commonly referred to as design flow. MakeBlock allows users to define a unique design flow at the beginning of each run, the action driven flow. In addition, event driven flow is supported where a design flow can be redefined in real time while a MakeBlock run is ongoing. This is frequently used for open-ended iterative repair loops when it is unknown as to how many passes will result in desired results. These two novel concepts to variable design flow will provide the extreme flexibility needed to leverage rapid prototyping.

1.3 Additional ASIC Flow Background

1.3.1 Past ASIC Design Cycle

The design cycle used for many years consisted of several distinct steps as shown in Figure 1 [18]. This design flow is very reactive rather than proactive in terms of violations and problems. Design issues were usually not uncovered until late in the design cycle and repair required numerous passes through the entire flow. Most of these repair cycles involved changes in RTL description of the system thus resulting in the entire design flow being repeated over, hence increasing design cycle time leading to large product delivery delays. Another problem present in this type of a design flow was associated with high performance designs with very tight timing constraints. Earlier stages in the flow, such as synthesis, did not comprehend any physical characteristics of a design in order to achieve timing closure. Instead, the entire front-end (RTL through floorplan) design was based on Wire Load Model (WLM), which is highly inaccurate. The WLM is based on approximation of delays, introduced by wiring through estimated capacitive loading, based on net fan-in and fan-out.

Figure 2: WLM vs Physical Synthesis corelates estimated parasitic effects using WLM and Physical Synthesis against the actual parasitic data obtained from post-layout parasitic extraction. As can be seen from
Figure 2, WLM synthesis is not only fairly pessimistic in estimating wiring capacitive parasitic effects, but in addition, it has very poor distribution correlation compared to desired or actual results. In other words, the WLM estimate of parasitics has almost no correlation with real parasitic effects. The WLM mostly relies on overestimation of these effects, which proved to be a functional approach on 0.18um and above technologies allowing timing closure without too large guard bands. However, below 0.18um this approach is too pessimistic and demands an extremely iterative block synthesis and hardening process in order to close timing.

Timing closure during synthesis gave no guarantees that timing will be met after placement and routing. This would result in many iterations of the cycle until timing was finally achieved, more through trial and error than design and implementation.

![WLM vs Physical Synthesis](image)

**Figure 2: WLM vs Physical Synthesis**

1.3.2 Present ASIC Design Cycle
The logic synthesis and block hardening methodology currently used in the VLSI industry shows a strong fusion between front-end (HDL design and synthesis) and back end (placement, routing and verification) [18] as can be seen in Figure 3. This new methodology is normally referred to as Physical Synthesis (PS) and its benefits can be seen in Figure 2. Even though the results are significantly pessimistic for physical synthesis, very similar to WLM, the distribution of physical synthesis results resembles a Gaussian distribution of desired or actual results.

Physical synthesis allows for most of the timing critical decisions to be moved forward in the design cycle thus shortening the cycle time and simplifying timing critical design hardening.
Fusion between front-end and back-end (floorplan design through verification) provided an ability to more accurately predict the true behavior of a design during earlier design stages such as synthesis. Physical synthesis tools, in addition to WLM based synthesis tools, take into account gate delays but also complete global and partial detailed routing in order to ensure more accurate parasitic estimation [18]. Routing capabilities, during synthesis, allow for silicon prototyping, thus ensuring accurate timing prediction and timing closure after the final routing stages.

However, it is still very difficult for one single EDA vendor to be the absolute leader in all areas of ASIC design and to produce a high quality single environment solution. For this reason, most design houses continue to leverage EDA vendor expertise in point tools and internally develop automation.

1.3.3 Future ASIC Design Cycle

Future tools of the EDA industry, for ASIC development, are clearly converging toward a single environment with an ability to quickly prototype a design. However, it remains to be seen if there will be a winning, high quality, single environment solution with enough flexibility and quick response to the needs of new deep sub-micron processes. Even if such a solution becomes available, other criteria such as cost, support, learning curve and acceptance by designers will most likely present a big problem. For these reasons, automation tools will continue to play a vital role in ASIC design for many years to come.

1.4 Definition of Terms

MakeBlock – software tool for automated block hardening

WLM – wire-load model
GDS – Graphic Design Station industry standard file format describing physical shapes on silicon

Tape-Out – term describing completion of a design and its shipment to mask manufacturer

IP – Intelectual Property

Soft IP – HDL described design

Hard IP – GDS or similarly described physical layout of a design

Block hardening – process of taking HDL and creating GDS representation of a design

HDL – hardware description language

Front-end – design flow steps from HDL creation to floorplan design and/or placement

Back-end – design flow steps from floorplan design and/or placement to physical and timing verification

VLSI Circuit – Very Large Scale Integrated Circuit

EDA – Electronic Design Automation

ASIC – Application Specific Integrated Circuit

RTL – Register Transfer Level (HDL code format)

RTL to GDS – process of taking RTL design to its physical representation

Synthesis – converting RTL design into gate level netlist

QA – Quality Assurance

PnR – Place and Route
TCL – Tool Command Language; interpreter and programming language

DEF – Design Exchange Format

LEF – Library Exchange Format

TLF – Timing Library Format

CTLF – Compiled Timing Library Format

SDF – Standard Delay Format

.lib – Liberty, Synopsys proprietary timing/power/functional file format

.cdl – Cadence proprietary noise library format

SI – Signal Integrity

IR analysis – analysis which takes into account supply voltage drop over and through interconnect. Analysis is calculated via ohms law, transient simulation by calculation of I*R to obtain voltage drop and is frequently called IR drop analysis.

DRC – Design Rule Check

CLDRC – Created Layer Design Rule Check

LVS – Layout vs. Schematic

RCX – Resistance and Capacitance Extraction

Latchup check – physical verification check ensuring circuit/chip is not susceptible to latchup; an event which causes large currents via effective pwr/gnd shorts under certain conditions such as high substrate resistance.
Soft check – Physical verification check ensuring no connections are established through high resistance materials such as wells, substrates and sometimes poly-silicon.

DFII – Design Framework II, Cadence proprietary design format for schematic and layout.

DSPF – Detailed Standard Parasitic Format


ECO – Engineering Change Order
CHAPTER 2. REQUIREMENTS

2.1 Scope

During this thesis, tools, methods, Quality Assurance (QA) suites and user training will be developed thus providing ASIC designers with an automated and flexible EDA tool. This tool will accept design and timing constraints as input and output physically clean GDS representation of the design with met timing requirements.

2.2 Customers

Customers of this thesis project include ASIC designers where all customers can be sorted into 3 categories:

Front-end experience – users with synthesis experience.

Back-end experience – user with placement and routing experience

No experience – user without Place and Route (PnR) experience

The final product will be aimed to be an asset for each of the user categories. By providing a Tool Command Language (TCL) based environment similar to synthesis tools, users with and without front-end experience can comfortably use and quickly come up to speed by using tool default values and flow making decisions. By providing a flexible and fully automated environment, users with back-end experience should gain significant speedup during the block hardening process. The final product must be flexible and intelligent enough not to burden and limit the ability of designers.

2.3 User Requirements

2.3.1 Functional Requirements
2.3.1.1 Input Requirements

2.3.1.1.1 Netlist

A (Verilog) netlist resulting from synthesis of a block must be provided. This netlist is created by a synthesis tool and should already contain a fully synthesized scan chain. The netlist should also contain model instances for any macros (e.g. embedded memories) in the block. MakeBlock must have access to the abstract layout of these macros for block level integration. Verilog netlist is required even for VHDL designs.

2.3.1.1.2 Floorplan (optional)

The preferred floorplan can be supplied in a Design Exchange Format (DEF) file format and should already contain placed cells with completed power grid and IO pins. The floorplan is optional as MakeBlock supports automatic floorplan generation via Cadence’s Silicon Ensemble and timing-driven placement via Qplace engine.

2.3.1.1.3 Library

One or more Library Exchange (LEF), Timing Library (TLF/CTLF), Standard Delay (SDF), Liberty (.lib), Noise (.cdl), etc format files will be accepted as input. These files should contain all information for generation of the physically correct layout including timing, antenna avoidance rules, antenna standard cell characterization, Signal Integrity (SI) noise standard cell characterization, power supply voltage (calculated via I*R transient analysis, hence frequently referred to as IR analysis) drop standard cell characterization, etc. In addition to standard cells, any macros such as RAM/ROM and analog blocks, information must be provided as required.

2.3.1.1.4 Block Level Constraints
All of the constraints that would be used during a manual pass through the block hardening process must be provided up front to MakeBlock. These constraints include the following:

Clock tree constraints - clock tree requirements such as skew, min/max delay, etc.

Timing constraints - timing requirements on any critical nets and list of false and multi-cycle path nets.

Power constraints - power usage restrictions.

Test constraints - test pin designations.

Signal Integrity constraints – cross-talk signal integrity thresholds and list of acceptable solutions to any signal integrity problems such as buffer insertion, routing changes, etc.

2.3.1.1.5 Configuration Settings

All of the configuration settings that would be used during a manual pass through the block hardening process must be provided up front to MakeBlock. These settings, which are provided by the user, include the following:

Special nets - any power/ground, clock, or test nets that have preferential routing priorities.

Routing parameters - routing settings (e.g. use off-grid routing).

Physical verification parameters - nets, metal options, cell list and other setting required for successful Design Rule Check (DRC), Created Layer DRC (CLDRC), Layout vs. Schematic (LVS), Resistance and Capacitance Extraction (RCX), Latchup and Soft checks.
Abstract parameters (optional) - parameters required to successfully complete abstract view generation in DFII using Envisia Abstract Generator.

Equivalency check parameters - parameters required for completing Conformal LEC runs on verilog netlists.

Design information - name and location of library in which to place the resulting layout. Also includes design name and stdcell library information.

Blockage specifications - blockages should be pre-placed to avoid routing over memories, analog blocks, etc.

Halo specification - any needed blockages around blocks to avoid inter-block DRC violations at block assembly time.

2.3.1.2 Output Requirements

The required outputs for MakeBlock are a timing and physical verification clean layout that meets all constraints and a final netlist, or a detailed explanation of why the flow failed and suggestions for correcting the problems.

2.3.1.2.1 Netlist

As part of the MakeBlock process, minor modifications may be made to the netlist due to insertion of buffers and other timing tuning mechanisms in addition to clock tree insertion and cross-talk signal integrity correction. The final netlist produced by MakeBlock must reflect the layout and must be functionally equivalent to the input netlist. Functional equivalency between netlists will be verified using Verplex Tuxedo.

2.3.1.2.2 Layout
The layout resulting from MakeBlock must pass all design verification tests (DRC, LVS, Latchup, Soft check, stress rules, etc). The layout must exist as a layout view in a Design Framework (DF) II database and will be preserved in GDS format if desired (a run option).

2.3.1.2.3 Abstract

MakeBlock will provide optional support for abstract view generation of the hardened blocks (option driven for flat designs), which will be created in the destination library of the block in DFII format.

2.3.1.2.4 Power

No power constraints or power optimization will be supported at this time.

2.3.1.2.5 Parasitics

Results of layout RC extraction in Detailed Standard Parasitic Format (DSPF) and Standard Parasitic Exchange Format (SPEF) will be preserved. Extraction will be performed during place and route for timing closure and on the final layout after abstract view substitution in the DFII environment. It is important to note that tools used at PnR and on the final layout differ and thus the final output may differ. In such case, a user will be required to rerun routing (probably an Engineering Change Order, ECO) with tightened requirements in order to meet timing.

2.3.1.2.6 Other

In addition to the primary output information listed above, other data that will be created by MakeBlock includes the following:

The final, routed DEF file - useful if an Engineering Change Order (ECO) is needed.
Reports - results of timing, power, area, and other verification runs condensed into some meaningful format. The reports should clearly indicate whether each constraint was met. For timing constraints, the reports should indicate the final timing margins.

ILM model - for top level timing analysis and top level clock insertion.

2.3.1.3 Process Requirements

2.3.1.3.1 Multi-Exit / Multi-Entry

Flow state should be saved after each major processing step in MakeBlock flow. This will facilitate rerunning a step or group of steps with a minimal amount of redundant processing.

2.3.1.3.2 Automation

Provided that all input information is correct and sufficient, MakeBlock should run to completion with no additional user input.

2.3.2 Environment and Hardware

MakeBlock must run in the standard corporate design environment which is currently Sun workstations running Solaris 8 or Linux Red Hat 7.3 operating systems.

Any code written for this system must conform to CAD Coding Standards and must be stored in the corporate CAD source repository. The CAD Coding Standards document will be updated, if deemed necessary, with TCL specific guidelines.

2.3.2.1 Programming Languages

A majority of this project will be written in the TCL/TK and C/C++ languages. Other languages such as shell scripting languages (sh,csh),
Skill (Cadence proprietary lisp derivative), Perl (www.perl.com) and Ruby (www.ruby-lang.org) will be used as well.

2.3.2.2 Operating System

The tool is targeted for Solaris 8 and Linux Red Hat 7.3 or higher operating systems.

2.3.2.3 Hardware

The tool is targeted for Sun sparc processors or x86 based workstations.

2.3.3 Performance and Capacity

2.3.3.1 Design Size Capacity

None, limited by vendor point tools.

2.3.3.2 Run Times

None, limited by vendor point tools.

2.3.3.3 Physical Memory

None, limited by vendor point tools.

2.3.3.4 Test Cases

The test case used during development, testing and validation is public IP, a 32 bit, low power processor core block. This design will be implemented in the 0.25um proprietary technology.
CHAPTER 3. ARCHITECTURE

The MakeBlock functional architecture, shown in Figure 4, was designed with special considerations in mind. It must be flexible enough to allow any point tools to fit within its environment while providing a seamless transition from one step to the next while managing all information about design, constraints and user feedback.

![Figure 4: MakeBlock Architecture](image)

The heart of the MakeBlock is its Design Flow Automation Engine, which is responsible for making decisions about the flow. It is responsible for
creating and modifying flow steps based on user input, various knowledge bases and real time events as they occur.

3.1 Knowledge Bases

Various knowledge bases are built into the MakeBlock and several different ways of adding knowledge bases are supported. Knowledge base refers to the particular tool or feature knowledge, which is stored and can be shared among users and projects. By providing knowledge base functionality, MakeBlock enables corporate learning and ensures learning occurred for one user or project is transferred elsewhere in the company. This system, as it grows, will enable new inexperienced users to create high quality hardened blocks as if they had years of experience using vendor point tools. All knowledge bases are accessible and modifiable during run-time, thus not forcing a user into a box. They try to be a guide along the right path.

3.1.1 Vendor Tool Knowledge Base

The Vendor Tool Knowledge Base is built with focus on EDA vendor point tools, which are external to the company deploying MakeBlock. This knowledge base includes information about what tools are available for solving particular problems such as Signal Integrity (SI), noise and delay violations. It also includes information about methodologies available within point tools that can be followed or used in order to identify and/or eliminate issues. In addition, bug workarounds and best practices are also stored and enforced via this system.

3.1.2 Design Knowledge Base

Corporate Design Knowledge Base is responsible for maintaining information about corporate design best practices, methodologies and procedures. Most of this information is proprietary and is left out of this document.
3.1.3 Technology Knowledge Base

The Technology Knowledge Base is used to store information about proprietary technology specifications. Through process features and descriptions, MakeBlock is capable of making many intelligent decisions and enforcing a range of rules, which would otherwise result in large delays as they are normally caught late in the design cycle.

3.1.4 Tool Knowledge Base

The Tool Knowledge Base is used to store information about internally developed tools and automation. This knowledge base ensures interoperability and compatibility with other internal tools in addition to enhancing reuse of Intellectual Property (IP). This base would include information about design management best practices, load distribution constraints in order to obtain run time speedup by running certain steps in parallel, etc.

3.1.5 Project Knowledge Base

The Project Knowledge Base is used to store specific information about the project. This information would include specific manufacturing processes the design is targeted for, specific tools, rules and constraints which apply to the entire project team, etc.

3.1.6 User Knowledge Base

The User Knowledge Base is used to store user specific properties, preferences, learning and constraints. By default, this knowledge base is empty but every user has the ability to update and maintain a personal knowledge base which is suitable for specific user tasks that can be automated.

3.2 Data Flow
The data flow for MakeBlock is modifiable and normally varies from one run to the next. The data flow is action and event driven by a user, run time events and results. For this reason, many different flows can exist and occur but default flow, which should meet requirements of a large percentage of designs is shown in Figure 5a: MakeBlock Default Data Flow.

Figure 5a: MakeBlock Default Data Flow
Figure 5a: MakeBlock Default Data Flow (continued)
PARASITIC EXTRACTION

To Floorplan  

ECO MODE

PASS=0  
TIMING<10  

TIMING MET

Y

N

STATICAL TIMING ANALYSIS

To SI ANALYSIS

SIGNAL INTEGRITY ANALYSIS

Figure 5a: MakeBlock Default Data Flow (continued)
Figure 5a: MakeBlock Default Data Flow (continued)
3.2.1 Endcap Cell Insertion

The PnR (see Figure 4) tool floorplans normally contain logic core area, input/output (IO) ring area and macro areas. For an example please refer to Figure 6.

![Full Floorplan Diagram]

**Figure 6: Full Floorplan**

The Logic Core Area is usually defined by rows, which can be used by placement tools to insert cells. Each row is usually the height of the standard cell height but it can also be a multiple of the standard cell height. Larger rows are used for specialized cell placement such as flip-flops, which requires more space but implementing it on a standard cell level requires too much area and performance is not guarantied. For
these reasons specialized larger cells are created in order to reduce die area and increase performance. Normally, smaller and larger rows do overlap each other in order to make area accessible for regular size standard cells as well as larger standard cells. Refer to Figure 7 for an example.

![Figure 7: Endcap Cells](image)

Endcap cells are used for terminating rows as the end of diffusion normally has special requirements such as latchup rules. In order to make layout DRC and electrically latchup compliant, endcap cells are created specifically for the reason of eliminating these errors. The tapping layer (N+ or P+), which is usually all that is contained in the endcap cell, does not interfere with routing, however, the endcap cell will block any other cells from being placed at that location.
3.2.2 Global Constraints Generation

The Global Constraints File (GCF) is a proprietary Cadence format, which specifies timing constraints associated with the particular design. These constraints are usually translated from synthesis constraints, along with timing library information, for the manufacturing process the design is targeted for. GCF constraints are similar to SDF, however they are more complete as they will include re-converging paths (Figure 8).

3.2.3 Clock Tree Insertion

The Clock Tree always requires special attention as this particular net has some very special requirements. Usually all sequential logic is triggered and driven by a single clock, which places a lot of capacitive load on the clock driver. This capacitive load is not only due to the intrinsic capacitance of sequential logic cells but is also due to the large interconnect capacitance. Sometimes clock trees are thousand of microns long resulting in very large capacitive loads and large clock skews at different points in sequential logic of a design. It is vitally important that all logic is operating synchronously in order to perform as intended. For this reason, clock tree insertion and balancing is performed by inserting buffers and inverters along the clock tree path and then the clock signal is routed through them. By inserting buffers, the capacitive load is reduced.
for the clock driver and performance of a design is increased as the entire chip can be clocked faster.

3.2.4 Filler Cell Insertion

Filler Cells are inserted after all cells have been placed and the confidence factor of a design in meeting timing is high. Filler cells will fill in all row spaces, which remain open. This step is very important as power rails (horizontal power lines) are usually built into the standard cells as feed-through. Leaving any space in the row would result in a break in the power line. Refer to Figure 9 for an example of filler cells.
3.2.5 Special Net Routing

Special nets include clock nets as well as any critical signals which require highly optimized routing paths (Figure 9). Critical signals are frequently routed in wider metal and are usually protected by power/ground wires running along its side in order to protect them selves or neighbors from noise caused by capacitive coupling with adjacent signals. For these reasons, special nets are routed first when the routing congestion is at its minimum.

3.2.6 Signal Routing

During this stage, all remaining signals defined and unconnected are fully routed (Figure 9). In modern generations of deep sub-micron processes, plasma etching is used during some of the manufacturing steps. Plasma etching involves a very high energy state of the matter which in turn can result in significant static charge build up on the circuits. This static charge will sometimes get high enough such that it will significantly damage or completely break down the gate oxide (Figure 10) by discharging through it into the substrate [19]. This can result in very unreliable devices or completely non-functional silicon [19]. Two basic methods exist for eliminating so called antenna violations: layer hopping and diode insertion.
Layer hopping is usually a preferred method and it consists of changing the current level of metal into the one above and back to the lower metal, thus reducing the total side area to gate area ratio. This method is preferred. It creates very little impact on timing, however, frequently not all antenna violations can be repaired using the layer hopping technique as routing congestion is significantly increased through this method.

Diode insertion is another viable method. It consists of insertion of diodes which are connected to the nets with a large side wall to gate area ratio. These diodes during manufacturing process, would be turned on in the forward bias mode by a static charge thus allowing the charge to be transferred into the wafer substrate without causing damage to the gate oxide. This method is very reliable and is fairly low cost in terms of area, but it can vary significantly affecting timing of critical signals as diode cells carry large intrinsic capacitance.

Modern routers utilize both methods, layer hopping at first until routing tracks are too congested, then diode insertion. Diode insertion is done through replacement of filler cells, which don’t have any functionality, thus not requiring placement legalization or re-routing.

Regardless of the router type, algorithms for routing can be divided into three stages. Global routing, detailed routing and finally search and repair. During search and repair, antenna violations are addressed and MakeBlock enforces 4 search and repair passes using the layer hopping
technique. Layer hopping is followed by diode insertion until all violations are eliminated or no significant progress is made in number of errors from previous to currently completed search and repair pass.

Routers can be divided into three possible categories, area routers, gridded routers and graph routers. MakeBlock is focused on gridded/graph routers, however it is fully capable of supporting area routers.

3.2.6.1 Area Routers

Area routers are the most flexible in terms of floorplan and other routing requirements. They are capable of creating routes anywhere in the available area and require very little user intervention. However, their capacity and performance is fairly limited. An example of an area router would include Virtuoso Custom Router (VCR) from Cadence Design Systems [8].

3.2.6.2 Gridded Routers

Gridded routers have much higher capacity, however they are not as flexible and require significant input from users. These routers are limited to creating routing paths only on a specified grid thus potentially costing area. Gridded routers trigger many other issues associated with the constant grid such as custom designed analog IP blocks. Custom blocks typically do not follow many of the PnR requirements and especially not some of the guidelines, which enable easier reuse of block for placement and routing. Gridded routers have very large capacities and are presently the most widely used in industry. An example would be Wroute by Cadence Design Systems [8].

3.2.6.3 Graph Routers
Graph routers are the new generation of gridded routers. They combine features of area and gridded routers to provide flexibility and ease of use along with large capacity and high performance. Graph routers dynamically establish virtual grids which the router can use thus providing all the benefits of area and gridded routers. An example of a graph router is Nanoroute by Cadence Design Systems [24].

3.2.7 Parasitic Extraction

Parasitic extraction is estimation of parasitic devices in the circuit thus enabling simulation of their impact on the circuit performance. Parasitic extraction is run on a gate/standard cell level rather than device level. Via this approach, extraction times are significantly improved without significantly sacrificing accuracy. Another benefit of using gate/standard cell level extraction is that extraction can be performed directly inside the PnR flow rather than having to convert it into GDS format. This significantly simplifies the number of tasks, eliminates potential triggering of tool bugs and reduces design cycle time. Output of the extraction is either a decoupled parasitic netlist in DSPF format, useful for static timing analysis, or a cross-coupled parasitic netlist in SPEF format useful for signal integrity analysis.

3.2.8 Static Timing Analysis

Static timing analysis is usually performed on a gate/standard cell level using the DSPF file from the parasitic extraction stage. During this stage, critical paths are analyzed for total delay, transition, setup and hold, and other critical timing parameters required by the design [17].

3.2.9 Signal Integrity Analysis

Signal Integrity Analysis is done on two levels. Firstly, glitch analysis is performed which identifies glitch violations where a 0/1 signal can be taken and stored as a 1/0 due to the noise effects from the adjacent lines
Figure 12: SI Effect on Signal Delay [10]

3.2.10 Signal Integrity Repair

Signal integrity (SI) repair can be done in three ways, buffer insertion, gate sizing and shielded routing. The preferred method for SI repair is a combination of buffer insertion and gate sizing [4]. The ECO command file contains a series of commands for the placement engine. This file requests that a buffer be inserted and gates sized on appropriate nets. The placement engine performs these tasks but routing needs to be completed again in order to reconnect newly inserted/sized gates. See Section 3.2.6 for details. Shielded routing is usually not done since buffer
insertion combined with gate sizing is sufficient. However, in some very extreme cases, critical nets can be routed with shielding, which causes the router to place power/ground nets on each side of the critical signal in the same level of metal. This reduces the cross-coupling capacitance and thus effects on the critical net, however, it is expensive in terms of area and routing grids.

3.2.11 DFII Generation

During this step, all data required for successful tape-out (TO), IP submission and archiving is imported in to the DFII environment. This environment is a Cadence proprietary environment and is used by range of tools to access required data. This process among others includes DEF to DFII-layout, verilog to DFfII-schematic and LEF to DFII-abstract conversion. Additional translators, such as DFII-layout to GDS layout, are also available and supported.

3.2.12 Physical Verification

Physical Verification is the final step where a design is verified for correctness from several view points. Firstly, the design is checked for compliance with manufacturing process and mask photo-lithography requirements via Design Rule Check (DRC). Secondly, the design is checked for latchup tolerance, electrical connections through high resistance materials (soft check) and layout vs. schematic (LVS) comparison. Finally, parasitic RC extraction and timing analysis are completed to ensure full timing compliance with the product requirements.
CHAPTER 4. IMPLEMENTATION

4.1 The Default Data and Flow Control

In a large user environment where collaboration on the same project is essential, the ability to configure and control default data and flow is extremely important. This ability must exist on several levels: a corporate, a particular geographic location, a specific designer team and an individual user. Coupled with modern design management, this collaboration becomes hard to manage. For this reason, a new corporate environment was created along with a range of default configuration files used by MakeBlock.

4.1.1 The Corporate Environment for Default Data Control

In order to create the most flexible environment that is portable across platforms, a set of files with an appropriate naming convention was established. This new set includes the following files:

- flow.env – used to setup values for all users and locations.
- tech.env – used to setup values for all users utilizing a specific technology and a specific manufacturing process.
- site.env – used to setup values for users of one geographic location.
- design.env – used to setup values for members of a single project. This can be applied to all users in the case of cross-site development, local users only or both.
- .userenv – used to setup values for a single user.

The files flow.env and tech.env are distributed via a Wide Area Network (WAN) to all corporate sites on a predetermined schedule along with the remainder of the corporate CAD flow. This ensures consistency between
all sites and all users. The file design.env is usually local to one site, but can also be distributed through some type of cross-site design management solution. In this case it affects all users assigned to a particular project regardless of their location. It can also be present at one site only and drive default values for designers of that project at that location. The files site.env and .userenv are completely local and respectively maintained by the system administrator and individual user.

This system provides the entire organization with the same behavior and feel of the software tools at all locations. Values from the corporate environment are accessed from different software tools through standard “envGetVal” procedures provided from the command line in Unix (C compiled executable) for scripting languages, and the Cadence SKILL DFII environment for SKILL programs. All files are processed in the order listed while returning the last found value. By specifying a property on the appropriate level, certain users are affected and knowledge is shared.

As an example, MakeBlock utilizes the flow.env file to setup a corporate knowledge base about the availability of point tools, thus defining available solutions for the MakeBlock flow.

4.2 The MakeBlock Environment for Data and Flow Control

Similarly to the corporate environment for default data control (4.1.1), MakeBlock supports a range of files allowing different levels of customization. The naming convention for MakeBlock is more uniform than that of the corporate environment. All files are named makeblock.tcl and can be located in any one of the five locations listed in Section 4.1.1. The MakeBlock user environment is TCL based, which reduces the development time because the TCL parser is utilized for syntax checking and the TCL language provides users with the ability to logically change the values and flow, based on algorithms. A typical example of a need for this functionality is the availability of multiple Silicon Ensemble (SE)
Several Silicon Ensemble executables, including *sedsm*, *seultra*, and *sesi*, can be run by the user or by MakeBlock. The cost of each of these tools is significantly different. Users should try to run the *sedsm* executable as much as possible, because it has the lowest price and typically has the highest feature count in the license pool. An algorithm can be written inside one of the makeblock.tcl files that can determine whether or not a user requires the advanced features of the *seultra* or *sesi* executables. Makeblock sets these executables as defaults for that particular run and block. This should significantly reduce the cost of the software pool required to tapeout a product.

Additional configuration files, that can be specified on the command line following the -config switch, are supported. These files are sourced subsequently to other files, but prior to the execution of any module specific procedures. The module specific procedures and data can be customized along with the MakeBlock procedures and data.

### 4.3 Information and Code Hierarchy

The TCL language, with its rapid prototyping and flexibility, was chosen for implementation. Performance was also considered and since all procedures are compiled prior to running, TCL 8.1 provided significantly better performance than the standard interpreter. In addition, TCL can easily be combined with a C/C++ compiled byte code for performance sensitive operations. Many other modules and utilities have been written in Cadence’s skill language, shell scripts, and C/C++ code, but the main architecture of MakeBlock was written in TCL.

The TCL language provides functionality to partition the code and data for complex projects, such as this one, through namespaces. TCL is not an object-oriented programming language and strongly resembles the C language. However, since the namespace functionality provides an ability to partition code and data, it achieves a pseudo object-oriented
appearance to developers. MakeBlock utilizes the namespace feature of TCL to generate a hierarchy of data and procedures.

Several namespaces supported by MakeBlock include:
Utilities namespace – used for shared procedures and methods.
MakeBlock namespace – used for MakeBlock procedures and data.
Module namespaces – used for module specific procedures and data.
Project namespace – used for design specific procedures and data.
Block namespace – used for block specific procedures and data.

Utilities, MakeBlock and module namespace are default namespaces used for procedure execution. In addition, project and block namespaces are supported. These namespaces can be populated by designers with project and block specific procedures and methods. MakeBlock first searches the block namespace, then the project namespace, for a predetermined set of procedures and, if found, executes them. Otherwise it defaults to the MakeBlock namespace procedure set. This is accomplished via the `makeblock::doProc()` procedure which provides users with an ability to fully customize the behavior of each of the modules. For details see Section 4.3.2.2.

4.3.1 The Utilities Namespace

Providing intellectual property (IP) is one of the most important tasks software developers can easily neglect when focusing on a specific project. However, by thinking outside the scope of the project, it is possible to recognize a range of procedures and functionality that can be developed to allow other unrelated projects to take advantage of newly developed code and a reduced development cycle. For this reason the `utilities::cy::cic:util` namespace was created. It is populated with a range of procedures and functions that can be shared among projects. This set of procedures includes global messaging, generic file manipulation functionality and generic client-server communication procedures.
4.3.2 The Top Level Namespace

The MakeBlock namespace is part of the Chip Integration Center (CIC) namespace, ::cy::cic::makeblock. In this namespace, data and procedures that are utilized by MakeBlock to collect and maintain data are defined. MakeBlock will contain multiple namespaces for each of the flow steps. For details refer to Section 4.3.3.

4.3.2.1 Data

One of the important considerations when creating data storage structures is potential user interface (UI) compatibility. TCL is tightly integrated with the TK library and provides rapid UI prototyping and development, while ensuring portability across multiple platforms. For this reason the data structure is a configuration array with all general MakeBlock settings. This array, in addition to the range of data required for MakeBlock, also includes the information passed in by the user and the intended job list for the flow.

4.3.2.2 Procedures

MakeBlock contains a range of procedures that are utilized by the individual tools to collect and modify data inside the flow. These procedures include a macro that defines the \texttt{run\_toolName()} set of procedures. This will correspond to the steps performed by MakeBlock as described in the job list. Each of the \texttt{run\_toolName()} procedures contains five calls (\texttt{init}, \texttt{prerun}, \texttt{run}, \texttt{postrun} and \texttt{status}) for each of the tool namespace procedures. For a detailed description see Section 4.3.3.

Each of these functions is executed through the \texttt{makeblock::doProc} procedure. This enables the user to override any default definitions or behavior for any of the module default flow procedures. This functionality provides the user with an easy way to redefine and customize all major tasks of the MakeBlock flow; however, the \texttt{makeblock::doProc} method
only operates on the procedures. All data is expected to be in the MakeBlock namespace. Other data in the project and the block namespace will be ignored by default.

Other utilities will be stored within this namespace that will provide a range of functionalities. The most important ones include: a procedure for setting up a module specific data configuration array, input and output file inheritance and maintenance, run directory creation and revision control, global messaging, and global flow control.

4.3.3 The Module Namespace

The data maintained in the tool namespace is information used for that particular tool’s run. In order to ensure that correct file information is passed from one step to the next, the \textit{init} procedure of the tool namespace will search the namespaces in reverse order than that specified by the MakeBlock job list. As soon as data is found, the search will stop and the data will be retrieved and stored into the current tool namespace. This process will ensure that the most recently modified and updated files are used for subsequent steps. It should never be the case that the search for data is empty. MakeBlock performs all data integrity verification at startup during the command line and file parsing. The new step should not be launched unless the previous step has been successfully completed and the correct output has been generated. Each module namespace will be a child of the MakeBlock namespace such as the \texttt{cy::cic::makeblock::"module"}.

Each of the tool namespaces will contain data pertinent to themselves, one command file string variable and five major procedures.

The \texttt{cy::cic::makeblock::module::init} procedure initializes tool variables. This procedure will be recursive in nature and will search the flow to get output from previously completed steps. It should never be the case that
the previous step does not have appropriate data, because MakeBlock shouldn't launch the next step without successful completion of the previous step. During this step the script or command file generation will be performed, and this information will be stored in a tool namespace configuration array. Via this mechanism, information will be available for modification by the user defined procedures prior to the input command file generation of the current task point tool.

The `cy::cic::makeblock::toolName::prerun` procedure generates the scripts and the command files required for the point tool run.

The `cy::cic::makeblock::toolName::run` procedure executes the tool with appropriate arguments. Execution of the appropriate command will be handled through the `envGetVal()` environment where all executables for each of the steps will be registered. During this step additional mb.replay files will be generated in the tool run directory for, if needed, future reruns of just this task. The run directory location for all tool runs will be determined by concatenating MakeBlock rundir with the tool run directory, thus requiring that the tool run directory be a subtree of the MakeBlock run directory or specified as a relative path with respect to the MakeBlock run directory. If redirection is desired, the user can use unix "link" executable to redirect the run location through symbolic links.

`cy::cic::makeblock::toolName::postrun` - process the run, and create any and all log files and information needed from it.

`cy::cic::makeblock::toolName::status` - return status of the run (0 for failure, anything else for success.)

All data required by the particular module is stored within the config() array of that module's namespace, which is very similar to top-level MakeBlock implementation.

4.3.4 Project and Block Namespace
As mentioned before in Section 4.3.2.2, project and block namespace is supported for procedures only. These namespaces will be searched for the appropriate function name (run_toolName) and, if found, will be executed rather than executing MakeBlock built-in procedures. The user can define in project/block namespace specific functions and customize any portion of it, either init, prerun, run, postrun or status.

4.4 Execution

4.4.1 MakeBlock Level Execution

At the top level, the MakeBlock execution sequence is as follows:

a) TCL shell launch with appropriate arguments – this step is done in order to avoid hard-coded full path to TCL shell executables. Via c-shell, TCL shell (tclsh) is launched with appropriate arguments, thus utilizing the first found tclsh executable in a user path.

b) Sourcing/loading of all modules – during this step all MakeBlock modules, both TCL and binary shared objects (.so), are loaded.

c) Command line parsing – the entire command line argument is parsed and data from it accumulated. Input is checked only for syntax and not data quality, which is performed later on. Command line is parsed in such a way as to allow for all arguments to be interpreted and all errors reported at once, rather than to exit on first error. This allows for quicker debugging and faster job setup time.

d) Configuration files are sourced – during this step all configuration files are sourced. First, the directory structure is searched for all makeblock.tcl files, which are sourced before any command line configuration files. Command line configuration files (-config option) are sourced last, thus having the final impact of the run settings.
e) MakeBlock initialization – during this step all design pertinent files are verified and processed, and all entry level module procedures are defined (run_toolName). While entry module procedures are defined, appropriate namespaces are searched for appropriate procedure definitions (see Procedures, Section 4.3.2.2). Needed file data is accumulated and stored in appropriate data structures and any errors or warnings are returned to the user. Depending on the severity of the problems, MakeBlock run will continue or abort.

f) Module execution – during this step, all modules listed in the task list are called and executed. MakeBlock expects return status from module runs and has no immediate knowledge of what a module task is. MakeBlock will also expect some data for a top-level log file, which is described in Reporting and Logging (Section 4.5). Before each of the modules are launched, the restart.cfg configuration file and mb.restart replay files are generated. This allows the user to restart run from the module where it terminated with an error. If an error is returned by one of the modules, MakeBlock terminates with appropriate messages and generates a top-level log file. If the status returned from all modules is OK, MakeBlock will continue running until the last module has been executed. The number of steps or modules executed is not limited and not set at any time throughout the MakeBlock run. The list of tasks, which MakeBlock will continue to execute until the end of the task list is reached, can be modified at any time. This feature enables full functionality of event driven flow enabling a user to go through numerous cycles until particular design constraints are reached, be they timing, area, power, etc.

g) Logging and cleanup – during this step, a top-level log file is generated, when restart and other files are removed.

4.5 Reporting and Logging
4.5.1 Messaging

For tools of high complexity it is extremely important to organize the central messaging mechanism. For MakeBlock messaging, the following format was chosen:

Severity: Message text … (MBLK-#)

Message severity can have one of three values: Info, Warn, or Error. This enables MakeBlock to determine if it is OK to proceed with or terminate the run, as well as provide an easy way to glance through log files and find problems.

Message number (MBLK-#) identifies what message was triggered and is hyperlinked in html files, thus allowing the user to easily access detailed documentation in case message text is not sufficient to determine the source of error.

The messaging system is capable of sending output to standard out, standard error or the file of the users choice by defining –log “filename” argument on the command line.

4.5.2 MakeBlock Log File

MakeBlock log file (mb.html) is the main log file for each run. The log file contains general information about the entire run and contains a small summary for a subtask with hyperlinks linking the user to the detailed summary for that subtask. An example of a MakeBlock top-level log file is shown in Figure 13.
4.5.3 Subtask Log File

Each of the modules contain their own log file named “stepName”.log. By default, they contain all of the detailed information the user needs to know about that particular run with the summary at the bottom of the log file. A special string for top MakeBlock log file, which will be used as a summary for the entire subtask run, will be stored in module namespace config(mblog) property, which is then collected by top-level logging procedures for the top-level log file. The value of this variable will be appended on the new line to the MakeBlock log file upon successful completion of the task. An example of a subtask log file is shown in Figure 14.

Figure 13: MakeBlock Top-Level Log File
4.6 Extending Functionality

The ability to modify, extend and add new functionality is one of the most important features of MakeBlock. Through user defined TCL script it is easy to gain access to any of the MakeBlock procedures and/or data structures. By defining a firm architecture of all modules (see section The Module Namespace, 4.3.3), a user can easily modify, add or remove functionality. Functionality alterations are done via configuration files written in TCL language (see section Configuration Settings, 2.3.1.1.5). Several levels of modification are possible, but they can all be broken down into:

a) Default module functionality alteration – includes all modifications of the default functionality, be they addition, modification or removal of functionality or data. In order to achieve any of these modifications, user defined procedures can be introduced into the appropriate execution namespace (see section Procedures, 4.3.2.2) thus controlling MakeBlock functional behavior or data structures. This method of functionality modification is most typical when the user relies on default MakeBlock behavior and functionality; then, with slight modifications, alters that behavior. As an example, the user would define the ::hotchip::ctrlr namespace and within declare the ::hotchip::ctrlr::"module"::init() procedure. Through the ::cy::cic::makeblock::doProc procedure, this newly introduced function
would be executed instead of the default 
::cy::cic::makeblock::"module"::init procedure. However, the user within the ::hotchip::ctrlr::"module"::init procedure can invoke the ::cy::cic::makeblock::"module"::init procedure and then append or modify data resulting from its execution, such as:

```tcl
proc ::hotchip::ctrlr::siggen::init {} {
  ::cy::cic::makeblock::siggen::init
  ::cy::cic::makeblock::exitIfErrors
  variable ::cy::cic::makeblock::siggen::config moduleConfig
  set moduleConfig(script) "DELETE WIRE clk1; \n $moduleConfig(script)"
}
```

Through this procedure, the user will delete all routed wires of the net "clk1" prior to routing the block.

b) New module addition – a complete new module can be added to the execution namespace via configuration files and a new module can be simply registered with MakeBlock by appending the new module to the ::cy::cic::makeblock::config(pflow) variable. As long as the new module is following the architecture defined in The Module Namespace, Section 4.3.3, the MakeBlock top-level will fully function with the new module as if it were internally developed.

4.7 API and TCL Extensions

One of the most crucial tasks is constant LEF and DEF file parsing. It is crucial that LEF/DEF file parsers are correct and very efficient. DEF files can easily be 200+ MB and continually have to be parsed after each module completes updates in order to maintain correct internal MakeBlock data structures. Correct information is critical as MakeBlock makes decisions based on these data structures. For this reason, Cadence Design Systems’ C/C++ implementation of LEF/DEF Application Programming Interface (API) with source code has been obtained from
4.7.1 TCL Extensions

The TCL language comes with TCL API, thus allowing a range of scripts to be written and used inside C/C++ programs. Likewise, TCL also comes with an extensive built in C library. There are two basic ways to extend functionality of TCL: First, building a new shell by compiling a completely new TCL interpreter with needed functionality; Second, and more common, is to create a shared object (such as .so for UNIX type Operating Systems or .dll for Windows) and dynamically load it as part of tclsh, wish or another TCL interpreter. TCL is equipped with a “load” command, which performs dynamic linking when shared objects are loaded such as:

    load /usr/local/lib/def5_4.so tcldef

The load command is followed by a string which is used to compute an initialization procedure, such as Tcldef_Init(). The load command is slightly cumbersome, as it requires a full path to be specified. On most UNIX Operating Systems, LD_LIBRARY_PATH can be used to define a shared object search path. In this case, the load statement can be relative such as:

    load def5_4.so tcldef

TCL provides another command package, which removes the burden from the users or developers to maintain shared object locations and paths. The package command provides for simple loading such as:

    package require tcldef => 5.4

In order to provide most flexibility and sharing between users and developers, we opted for dynamically linked shared objects. After
compiling Cadence’s LEF/DEF API, we created wrappers for all procedures using SWIG 1.1. SWIG [20] is a software development tool used for the rapid building of C/C++ interfaces for various scripting languages such as: Python, Ruby and TCL. Along with LEF/DEF API, some basic perl and shell scripts were provided which quickly helped create procedure wrappers for all LEF/DEF API C/C++ functions and classes.

4.8 Supporting Implementation

In addition to the main working modules listed in Data Flow, Section 3.2, a significant number of modules and supporting implementations had to be put in place in order to fully automate the block hardening flow.

4.8.1 DFII Environment Tools

A range of dfII environment tools have been developed to aid the user during design, as well as to enable full automation of the block hardening process.

4.8.1.1 Abstract Generator Layout Compliance Tool

One of the basic requirements for Block Based Design Methodology (BBDM) and reuse is solid IP generation. Creating LEF representation for PnR modules is a fairly straightforward process; however, creating LEF representation of custom and analog blocks is significantly more difficult. In order to reduce cycle time impact on designers, AbGen Layout Compliance Tool was created which verifies the range of rules and guidelines on custom layouts. These rules are shown in Error! Reference source not found. and they include:
Figure 15: Abstract Generator Compliance Checker

a) The existence of net and boundary purposes for all routing layers is verified.

b) DFII libraries are verified for correct layer sets in accordance with the process specifications.

c) The top-level cell is verified for existence of a single prBoundary:drawinig layer, and it is correct with respect to the entire area of the cell.
d) All physical pin shapes are verified for a single instance in the layout. If multiple shapes are found, the user is instructed to ensure its accuracy and to use appropriate control variables to ensure these pins are connected by the router (mustJoin). This case is very common for cell/IP feed-through connections.

e) All pin shapes are verified to be on low resistance metals. Higher resistance metals, such as local interconnect, are flagged with warnings. Feed-throughs on high resistance metals adversely affect timing and are strongly discouraged.

f) All pin shapes are verified to be on the technology allowed routing layers.

g) All pin shapes are verified for minimum allowed features, such as minimum width, as required by the manufacturing process.

h) All pin shapes are verified to be of a rectangular shape and warnings are issued if pins are not square.

i) All pin shape centers are verified for compliance with the manufacturing grid, both vertically and horizontally.

j) All centers of drawing purpose shapes, overlapping pin shapes, are verified for compliance with the manufacturing grid, both vertically and horizontally.

4.8.1.2 Connectivity Checker

Frequently, minor alterations of the final physical layout are needed. In order to make this possible inside the Virtuoso DFII environment, while still maintaining the ability to create DEF format from DFII database, certain criteria must be met:
a) All drawn paths must be of the object “path” -- not rectangle or polygon.

b) All contacts must be programmable cells (p-cells) exactly equal to LEF default via contacts.

c) All shapes must have established connectivity or reference the net they belong to.

In order to make requirement C easy to handle, Connectivity Checker was created which speeds up the connectivity association with newly drawn shapes by:

a) Identifying shapes that do not have connectivity.

b) Providing a user with a quick and easy-to-use Graphic User Interface (GUI) used to create connectivity on newly drawn shapes as shown in Figure 16. The user would select a drawn object, type a net name into the form, and apply it. The program would then correlate the net name with database nets and create properties on the selected object.

Figure 16: Connectivity Addition Form

4.8.1.3 DEFin Repair

Upon completion of the DEF file import into the DII database, layout is shown as it was described in the DEF file. Due to the differences between custom and PnR flows, some layout alteration and manipulation must be performed inside the DII environment. In order to provide that MakeBlock is fully automated, as well as to reduce designer cycle time,
the DEFIn Repair tool was written. Repair steps performed are shown in Figure 17 and they include:

a) Open cell using layout – modifies some of the cell properties to force opening of the cell with Virtuoso Layout Editor rather than Virtuoso Floorplanning tool.

b) Change abstract->layout – modifies all cell instances from their abstract representation into their physical layout. This enables full accuracy of physical verification runs.

c) Copy all metal pin shapes to drawing – for each shape of purpose “pin”, a new shape of purpose “drawing” will be created. This ensures DRC rules are met for pin enclosure by metal shape.

d) Fix bus net names – modifies all bus net names by changing all square brackets ([, ]) into buss notations supported by DFII or less than(<)/greater than(>) symbols.

e) Fix changed layer to fit layout – modifies size of the changed layer to fit a physical cell boundary.

f) Create labels on pins – creates text labels on pins in order to enable LVS to be run on the layout. In addition, this helps designers debug any LVS issues.
4.8.2 Technology Implementation

Some of the most important information any CAD flow can maintain is technology implementation data. This data includes all information required to describe the manufacturing process for which the design is targeted. This data includes: electrical parameters, such as sheet resistances for all layers; contact resistances of all vias; and parasitic information for layers and devices; etc. This information is used by CAD tools at various points to perform accurate calculations, such as RC parasitic layout extraction, or estimates, such as RC parasitic schematic extraction.

In order to enable MakeBlock to function properly with a series of vendor tools, new information had to be supported. In order to ensure the quality and ease of maintenance of this data from one technology process to the next, a significant amount of automation has been implemented; thus learning is captured and transferred to other CAD engineers in the corporation.

4.8.2.1 Programmable Cells
P-cells [16], are programmable layout cells which, based on user input, change and modify their appearance, and thus alter functionality and performance, or change other aspects of a device or circuit. The most frequently used structures and shapes are usually automated as p-cells, such as: devices, contacts, and rings. For an example of a p-cell, refer to Figure 18, which shows a typical implementation of a metal 1 to metal 2 contact array. In this particular case, an array of 4x3 vias was chosen as one of the properties. Refer to Figure 19.

Figure 18: Met1-Met2 Contact Pcell
Upon completion of the placement and routing of a block, the design is usually brought into the DFII Cadence environment for final verification and inspection. In order for the DEF-in process to be successful, DFII
technology data is required to have a predefined set of structures, which match technology information used by PnR tools.

4.8.2.1.1 Contacts

a) Special Net Contacts

Special net main busses, such as power and ground nets, are usually created during floor plan creation. Connecting cell power and ground nets to the main busses requires special attention in order to avoid IR drop related issues. Typically, special net main busses are much wider than signal wires. In order to reduce resistance and thus eliminate potential power IR drop problems, special net router [8] defines maximum size via arrays, which can fit within the overlap area of two shapes, as shown in Figure 20 [8]. A special net router somewhat inefficiently defines a via for each connection of power wires [13]. However, once the DEF data is introduced into the DFII environment, it can be efficiently represented with a single special net via which can be instantiated multiple times with different properties, thus molding to the specific location where used. Virtuoso requires that the p-cells used for special net vias be named ruleVia, and have either a defined specific set of properties, or be declared the via symContactDevice() [16] p-cell class, which defines a p-cell with the same set of required properties. The preferred way of defining rule-viases is to use the ruleContactDevice() [16] Virtuoso SKILL procedure, as it significantly simplifies and shortens implementation.
b) Non-Default Contacts

Non-default rules (see Section 4.8.2.2 for further details) support non-default rule specific contacts. In order to ensure that proper vias are used for connections between non-default signal wires inside the DFII environment [16], proper p-cells (see Section 4.8.2.1 for more details) must be defined. In addition to being defined, these cells have to be registered with Virtuoso as contacts to be used during the DEFin process via prViaTypes() procedure [16].

4.8.2.2 Non-Default Rules

Most of the routing performed on ASICs uses minimum width and minimum spaced wiring. Sometimes, critical nets require larger than minimum width and/or larger spacing from other nets in order to reduce resistance and/or cross-talk glitch/delay issues respectively. Through these methods timing can be improved and functional failures eliminated. The LEF/DEF syntax [13] supports this type of functionality by providing non-default rules, which can be defined as part of the LEF files and used on specific critical nets. In order for non-default wires to be properly imported into the DFII database, proper support must be defined as part of the DFII technology file [16]. Skill procedures have been created to expedite creation and simplify the maintenance of these rules [15]. The SKILL procedures defined accept: the non-default rule name, the multiples of minimum wire widths used to determine the widths of non-default wires, the lists of metal layers that the rule applies to, and the list of vias that should be used with the rule. This is very important as non-default rules frequently include more than just a single via for connections between different levels of metal due to the wide wire size (see Figure 21 for an example). By providing a via array, rather than a single via contact, resistance can be significantly reduced (4x as shown in Figure 22). The creation of via contact p-cells has been automated through p-cell classes
compliant with symContactDevice and/or the use of ruleContactDevice()
Virtuoso SKILL procedure.

Figure 21: Non-Default Rule Via Contacts
4.8.2.3 LEF Technology Compilation

Most of the process technology information is kept in the special SKILL language based environment, which is then used to compile data into a Virtuoso compliant ASCII technology file data format. In order to ensure equivalency between the technology process data used by the back-end set of tools, such as Virtuoso and Assura, with the PnR set of tools, both sets of technology data had to be created or compiled from the same source. In order to avoid creation of an external compiler and to convert the existing SKILL technology representation into LEF format, the Virtuoso environment has been used to import a technology file [16] and build a library, then dump the same library information into an LEF file through the LEFout process. This process has been fully automated and scheduled for nightly builds to ensure matches between back-end physical verification and PnR technology data. However, the downside to this implementation is that information compiled into an LEF file has to be
represented and supported in the Virtuoso technology file. If the Virtuoso
technology representation doesn’t support new features, such as antenna
information and rules, the LEF file will not contain this information. For
this reason, it is common to find several LEF files associated with a
particular manufacturing process. Most of the information in an LEF file is
automatically generated, but some information is entered and maintained
manually due to a lack of functionality inside the Virtuoso technology files.

4.8.2.4 DPUX Technology Compilation

The DPUX file is very similar to the LEF files, but was independently
created for Physical Design Planner (PDP) tools, which were later
acquired by Cadence [14]. As part of PDP, the Envisia Abstract
Generator [14] is provided, which enables the creation of the LEF files
from GDS and the technology (DPUX) file. This enables designers to
create an LEF representation for custom and analog blocks, which can
later be reused from the IP library for new design projects.

The DPUX file is created through a LEF to DPUX compilation process.
For this, executable lef2dpux is used, which is part of the PDP installation
tree. A range of modifications to layers and GDS maps had to be made in
order to ensure compatibility between GDS files and the Envisia Abstract
Generator.
Prior to launching a project, a preliminary analysis is done to determine Return on Investment (ROI). Usually corporations are looking for large ROI (usually higher than 5), in order to determine if the project is profitable and should be launched. It was determined that the MakeBlock ROI was around 10 across the broad user spectrum, such as novice to expert. However, for the purposes of this thesis we will only compare design cycle improvement or speedup for the case of an expert engineer familiar with design and tools used to harden the block. For this experiment, the computer system used to run MakeBlock on was a Sun Microsystems, Ultra 80 with dual SPARC II processors and 4GB of Random Access Memory (RAM), running Solaris 2.8 Operating System (OS).

For experimentation and testing purposes, a design from public, open source Intellectual Property (IP) has been chosen. The chosen design was a nnARM, 32-bit RISC microprocessor core, very similar to the ARM7 family of microprocessors designed and sold as intellectual property by one of the top commodity RISC architecture IP suppliers in the world, Advanced RISC Machines Ltd. (ARM) [23]. The nnARM processor core is a low-power 32-bit RISC microprocessor core optimized for low power and low cost. It offers about 100MIPs and is very suitable for use in personal audio players, wireless handsets and PDAs, pagers, low cost inkjet-jet/bubble-jet printers, and digital cameras [23].

Typical silicon design can be optimized for different parameters, which are usually mutually exclusive. These parameters are the speed of the design (frequency), power consumption and silicon area used. In order to reduce silicon area, usually frequency is sacrificed. On the other hand, if frequency is to be increased, this usually leads to higher power consumption and larger silicon area. For the purposes of this research we decided to focus on silicon area reduction.
During this stage, expectation is to achieve more than 10% silicon area reduction while meeting timing requirements. In addition, full block hardening process must be fully automated with easy to read and easy to understand output.

5.1 Choosing Test Case - nnARM Core

Intellectual property (IP) is one of the hottest and most highly coveted commodities in the new economy. Nowhere is this more apparent than in the semiconductor arena, where an IP revolution is defining a new generation of digital electronic products. IP is forming the basis of today’s ever-advancing microprocessors and is driving innovation across a broad spectrum of consumer and business applications -- pushing high-speed, high-bandwidth communications and wireless connectivity to new limits.

The use of microprocessors in a wide range of electronic devices has escalated to the extent that the use of microprocessor IP is now ubiquitous in system-on-chip (SoC) designs, providing the technology foundation for nearly everything electronic in the world today. With enormous growth potential, the IP market has emerged as one of the most dynamic sectors of the high-technology industry. Andrew Allison, an industry analyst, reports that in 2001, more than 538 million RISC (Reduced Instruction Set Computing) microprocessors were shipped, 74.6% of which were based on the ARM® microprocessor architecture. Growth in the IP market continues to be driven by the demand for semiconductor devices, which the Semiconductor Industry Association reports reached $139 billion in global sales in 2001.

Advanced RISC Machines Ltd. (ARM) was created in 1990 in a joint venture funded by Apple Computer and VLSI Technology. In 1991 ARM designed the first low-cost RISC architecture and soon thereafter in the early nineties, pioneered the concept of openly licensable IP for development of 32bit RISC microprocessor based SOCs. By licensing
rather than manufacturing and selling its chip technology, ARM established a new business model that has redefined the way microprocessors are designed, produced and sold. However, rather than purchasing IP, an open-source ARM (ARM7) like core is publicly available for download at [22]. In this we opted for the free version, which was designed with the following characteristics[23]:

- Up to 130 MIPs performance on a typical 0.13µm process
- Small die size and very low power consumption
- High code density, comparable to 16-bit micro-controller
- Wide operating system and RTOS support - including Windows CE, Palm OS, Symbian OS, Linux and market-leading RTOS
- A wide choice of development tools
- Simulation models for leading EDA environments
- Excellent debug support for SoC designers
- Multiple sourcing from industry-leading silicon vendors
- ARM instruction sets
- A three-stage pipeline
- Unified bus architecture
- A low power, fully static design
- A small die size
- Coprocessor interface
- A synthesizeable design
5.2 ARM7 Organization and Architecture

ARM7 organization and architecture is shown in Figure 23 [23].

![ARM7 Organization and Architecture](image)

Figure 23: ARM7 Organization and Architecture [23]

5.3 ARM7 Instruction Set

Summary of the ARM7 instruction set is sown in Figure 24 [23].
5.4 Test Case Preparation

In order to complete testing, several steps had to be performed. Data had to be synthesized, meaning, a Register Transfer Level (RTL) design had to be compiled into a gate verilog netlist, physical constraints had to be described through a floor plan design, and timing constraints had to be specified.

5.4.1 Design Synthesis

The obtained RTL code was synthesized into a gate level netlist using Synopsys’ Design-Compiler software on a proprietary 0.25u, 4
interconnect metal layer process and using proprietary standard cell libraries. The design is fully soft IP, meaning no macros or larger components than combinational logic gates and flip-flops were used.

5.4.2 Floor Plan Design

The floor plan was designed using Cadence’s Silicon Ensemble. Power stripes and rails have been arbitrarily positioned with the idea to reduce IR drop effects on the design.

5.4.2.1 IO Requirements

No special requirements have been considered as part of the IO pin placement other than limiting pin placement on west and east side of the die. All IO pins have been implemented in the top metal layer.

5.4.2.2 Area Requirements

The design was targeted for a 3.5 x 3.8mm area with an approximate utilization of 60%. Given no hard macros for the given design, utilization could have easily been increased, but since this experiment was first passed through a design hardening process, plenty of space was left to ensure the design’s timing closure. See Figure 25 and Figure 26 for details.
5.4.2.3 Power Requirements

No particular power/ground considerations were needed given the design's small size. Two vertical straps and two horizontal rails have been created on the third and the second level of the metal, respectively, thus ensuring an adequate supply to the entire design. See Figure 26 for details.
5.4.3 Timing Constraints

5.4.3.1 Clock Timing Constraints

Via clock timing constraints -- the speed of the design was targeted for 200MHz. This was achieved by setting the clock period to 5ns via following Synopsys' Design/Physical Compiler command:

```
clock -name CLK -period 5.000000 -skew 0.05 -waveform {0.0 4.5}
```
5.4.3.2 Signal Timing Constraints

No particular signal constraints have been created other than input and output delays based on the clock period via following commands:

\[
\text{set\_input\_delay [expr 0.33 * $clk(-period)] -clock CLK [all\_inputs]}
\]

\[
\text{set\_output\_delay [expr 0.33 * $clk(-period)] -clock CLK [all\_outputs]}
\]

5.5 Design Hardening

In order to completely harden the nnARM Core design, the following MakeBlock modules will be exercised:

- ctgen – clock tree insertion and balancing. Via buffers and inverters clock skew and delay balance throughout the design.

- eqc – equivalency checking. In order to ensure buffer/inverter insertion during the ctgen stage doesn’t alter functionality, the eqc step was run.

- ecgen and fcgen – end-cap and fill-cell insertion to ensure proper physical verification requirements for drc and latchup.

- snrgen and siggen – special net and signal routing.

- rcgen – parasitic RC extraction to produce coupled SPEF output.

- def2ic – import final DEF database into dfll environment.

- vloc2ic- import final verilog netlist into dfll environment.

- pv – run entire physical verification suite.

- ic2gds – create GDS design database.
- abgen – create an abstract for a design in form an LEF file for IP submission.

- lef2ic – finally, import LEF file into dfll database.

The MakeBlock command line arguments (as stored in mb.replay file) are as follows:

```
RegisterFile -def io/def/inst_RegisterFile.def -lef io/lef/slca00_r52t.lef -gcf 
io/gcf/temp.gcf -vlog io/vlog/inst_RegisterFile.v -lib mb_ram52t3 -config 
mb.cfg -log mb.log -flow ctgen eqc ecgen fcodegen snrgen siggen rcgen 
def2ic vlog2ic pv ic2gds abgen lef2ic
```

The MakeBlock configuration input files have the following content:

```bash
## ctgen
set ::cy::cic::makeblock::ctgen::config(constFile) (io/ctgen/RegisterFile.const)
## eqc
set ::cy::cic::makeblock::eqc::config(goldenLibList) "io/vlog/std/stdlib.module"
set ::cy::cic::makeblock::eqc::config(revisedLibList) "io/vlog/std/stdlib.module"
## snrgen
set ::cy::cic::makeblock::snrgen::config(pwrRtCmd) "SROUTE"
set ::cy::cic::makeblock::snrgen::config(pwrLayer) "met1"
set ::cy::cic::makeblock::snrgen::config(pwrLayerWidth) "4"
set ::cy::cic::makeblock::snrgen::config(sNetList) "@CLOCK"
## rcgen
set ::cy::cic::makeblock::rcgen::config(simplexLibDir) "io/simplex/slca00_r52t"
set ::cy::cic::makeblock::rcgen::config(simplexTechFile) "io/simplex/slca00_r52t.cl/ram52t-3.tch"
set ::cy::cic::makeblock::rcgen::config(spefout) "io/rcx/nnARMCore.spef"
## ic2gds
set ::cy::cic::makeblock::ic2gds::config(gdsout) "io/gds/RegisterFile.ic2gds.gdsout"
## vlog2ic
set ::cy::cic::makeblock::vlog2ic::config(refLibs) "tech, slca00_r52t"
```
5.6 Final Results

MakeBlock produces 2 sets of outputs -- first is ASCII via stdout, and a second via log files in html format.

5.6.1 Stdout Output

By default, generic information about the run is displayed via stdout with a user having the ability to redirect that output into an ASCII file of choice.

For the nnARM Core run, the following was the captured output:

Generic output with tool version and input files passed by the user:

```
Info: Version: 1.0  (MBLK-20)
Info: mb.cfg: processing file ...  (MBLK-4)
Info: mb.replay: writing file ...  (MBLK-6)
Info: io/vlog/inst_RegisterFile.v: processing file ...  (MBLK-4)
Info: io/lef/slca00_r52t.lef: processing file ...  (MBLK-4)
Info: io/def/inst_RegisterFile.def: processing file ...  (MBLK-4)
Warning: LEF with 5.4 and DEF with 5.2 VERSION mismatch ...  (MBLK-49)
```

Ctgen module launch for clock-tree synthesis (see Figure 5a):

```
Info: Launching ctgen step...  (MBLK-27)
Info: ctgen::config(vlogout): File name not defined, setting to io/vlog/inst_RegisterFile.v.ctgen.vlogout ...  (MBLK-34)
Info: ctgen::config(defout): File name not defined, setting to io/def/inst_RegisterFile.def.ctgen.defout ...  (MBLK-34)
Info: mb.restart: writing file ...  (MBLK-6)
Info: restart.cfg: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/ctgen: directory created.  (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/ctgen/ctgen.cmd: writing file ...  (MBLK-6)
Info: Executing command: "ctgentool ctgen.cmd >& ctgen.log" ...  (MBLK-10)
Info: ctgen.html: writing to log.  (MBLK-8)
Info: ctgen: Completed ...  (MBLK-29)
```

73
EqC module launch for equivalency check ensuring gate level netlist is still functionality equivalent after clock-tree synthesis (see Figure 5a):

Info: Launching eqc step...  (MBLK-27)
Info: 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/io/vlog/inst_RegisterFile.v.ctgen.vlogout: processing file ...  (MBLK-4)
Info: mb.restart: writing file ...  (MBLK-6)
Info: restart.cfg: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/eqc: directory created.  (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/eqc/eqc.cmd: writing file ...  (MBLK-6)
Info: Executing command: "lec -nogui -dofile eqc.cmd >& /dev/null" ...  (MBLK-10)
Info: eqc.html: writing to log.  (MBLK-8)
Info: eqc: Completed ...  (MBLK-29)

ECgen module launch for end-cap cell insertion (see Figure 5b):

Info: Launching ecgen step...  (MBLK-27)
Info: ecgen::config(defout): File name not defined, setting to 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/io/def/inst_RegisterFile.def.ctgen.defout.ecgen.defout ...  (MBLK-34)
Info: mb.restart: writing file ...  (MBLK-6)
Info: restart.cfg: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/ecgen: directory created.  (MBLK-22)
Info: 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/ecgen/ecgen.cmd: writing file ...  (MBLK-6)
Info: Executing command: "sedsm -gd=ansi -j=se.jnl -m=250 "EXECUTE ecgen.cmd ; FQUIT ;" >& ecgen.log" ...  (MBLK-10)
Info: ecgen.html: writing to log.  (MBLK-8)
Info: ecgen: Completed ...  (MBLK-29)
Info: 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/io/def/inst_RegisterFile.def.ctgen.defout.ecgen.defout.ecgen.defout: processing file ...  (MBLK-4)
Info: Launching fcgen step...  (MBLK-27)

FCgen module launch for filler cell insertion (see Figure 5b):

Info: fcgen::config(defout): File name not defined, setting to 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/io/def/inst_RegisterFile.def.ctgen.defout.fcgen.defout ...  (MBLK-34)
Info: mb.restart: writing file ...  (MBLK-6)
Info: restart.cfg: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/fcgen: directory created.  (MBLK-22)
Info: 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/fcgen/fcgen.cmd: writing file ...  (MBLK-6)
SNRgen module launch for special net routing (see Figure 5b):

SIGgen module launch for signal net routing (see Figure 5b):
RCgen module launch for parasitic interconnect RC extraction (see Figure 5c):

Info: Launching rcgen step... (MBLK-27)
Info: config(dspfout): File name not defined, setting to rcgen/RegisterFile.dspf ... (MBLK-34)
Info: rcgen: directory created. (MBLK-22)
Info: rcgen::config(layoutScale): parameter not defined, setting to ... (MBLK-44)
Info: mb.restart: writing file ... (MBLK-6)
Info: restart.cfg: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/rcgen/rcgen.cmd: writing file ... (MBLK-6)
Info: Executing command: "qx -nm -cmd rcgen.cmd
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/io/def/inst_RegisterFile.defctgen.defout.ecgen.defout.fcgen.defout.snrgen.defout.siggen.defout >>& /dev/null" ... (MBLK-10)
Info: rcgen.html: writing to log. (MBLK-8)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/rcgen/rcgen.log: processing file ... (MBLK-4)
Info: rcgen: Completed ... (MBLK-29)

DEF2IC module launch for converting DEF format to DFII (see Figure 5c):

Info: Launching def2ic step... (MBLK-27)
Info: mb.restart: writing file ... (MBLK-6)
Info: restart.cfg: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/def2ic: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/def2ic/def2ic.cmd: writing file ... (MBLK-6)
Info: Executing command: "icfb -nograph -replay def2ic.cmd -log def2ic.log >>& /dev/null" ... (MBLK-10)
Info: def2ic.html: writing to log. (MBLK-8)
Info: def2ic: Completed ... (MBLK-29)

VLOG2IC module launch for DFII schematic creation (see Figure 5c):

Info: Launching vlog2ic step... (MBLK-27)
Info: mb.restart: writing file ... (MBLK-6)
Info: restart.cfg: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/vlog2ic: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/vlog2ic/vlog2ic.cmd: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/vlog2ic/vlog2ic_opus.cmd: writing file ... (MBLK-6)
Info: Executing command: "ihdl +NOXTRSCH +DUMB_SCH - IGNOREEXTRAPINS -VERBOSE -param vlog2ic.cmd /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/io/vlog/inst_RegisterFile.v.ctgen.vlogout >>& vlog2ic.log" ... (MBLK-10)
Info: Executing command: "icfb -nograph -replay vlog2ic_opus.cmd -log vlog2ic_opus.log >>& /dev/null" ... (MBLK-10)
Info: vlog2ic.html: writing to log. (MBLK-8)
Info: vlog2ic: Completed ... (MBLK-29)

PV module launch for physical verification (see Figure 5c):

Info: Launching pv step... (MBLK-27)
Info: config(schName): parameter is set to RegisterFile (MBLK-31)
Info: config(schLib): parameter is set to mb_ram52t3 (MBLK-31)
Info: config(drcSwitch): parameter is set to fine_pitch (MBLK-31)
Info: config(clcSwitch): parameter is set to waffle_chip fine_pitch (MBLK-31)
Info: config(clcMaskAdd): parameter is set to plm li1m (MBLK-31)
Info: config(clcMaskDrop): parameter is set to plm li1m (MBLK-31)
Info: mb.restart: writing file ... (MBLK-6)
Info: restart.cfg: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/drc: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/stress: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/clc: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/latchup: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/soft: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/lvs: directory created. (MBLK-22)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/drc/vdrc.replay: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/drc/run.tcl: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/stress/vstress.replay: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/stress/run.tcl: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/clc/vclc.replay: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/clc/run.tcl: writing file ... (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/latchup/vlatchup.replay: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/latchup/run.tcl: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/soft/vsoft.replay: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/soft/run.tcl: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/lvs/aext.replay: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/pv/lvs/run.tcl: writing file ...  (MBLK-6)
Info: drc: Launching job ...  (MBLK-45)
Info: Executing command: "tclsh run.tcl 157.95.47.23 39031 &">& pv.log &" ...  (MBLK-10)
Info: stress: Launching job ...  (MBLK-45)
Info: Executing command: "tclsh run.tcl 157.95.47.23 39031 &">& pv.log &" ...  (MBLK-10)
Info: cldrc: Launching job ...  (MBLK-45)
Info: Executing command: "tclsh run.tcl 157.95.47.23 39031 &">& pv.log &" ...  (MBLK-10)
Info: latchup: Launching job ...  (MBLK-45)
Info: Executing command: "tclsh run.tcl 157.95.47.23 39031 &">& pv.log &" ...  (MBLK-10)
Info: soft: Launching job ...  (MBLK-45)
Info: Executing command: "tclsh run.tcl 157.95.47.23 39031 &">& pv.log &" ...  (MBLK-10)
Info: lvs: Launching job ...  (MBLK-45)
Info: Executing command: "tclsh run.tcl 157.95.47.23 39031 &">& pv.log &" ...  (MBLK-10)
Info: 1: Finished job with status 1 ...  (MBLK-46)
Info: 2: Finished job with status 1 ...  (MBLK-46)
Info: 3: Finished job with status 1 ...  (MBLK-46)
Info: 4: Finished job with status 1 ...  (MBLK-46)
Info: 5: Finished job with status 1 ...  (MBLK-46)
Info: 6: Finished job with status 1 ...  (MBLK-46)
Info: pv.html: writing to log.  (MBLK-8)
Info: NORMAL: writing to log.  (MBLK-8)
Info: pv: Completed ...  (MBLK-29)

IC2GDS module launch for conversion of drawn DFII to GDS (see Figure 5c):

Info: Launching ic2gds step...  (MBLK-27)
Info: mb.restart: writing file ...  (MBLK-6)
Info: restart.cfg: writing file ...  (MBLK-6)
Info: /home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/ic2gds: directory created.  (MBLK-22)
ABgen module launch for abstract and LEF generation (see Figure 5c):

LEF2IC module launch for conversion of LEF to DFII abstract view (see Figure 5c):
Info: 
/home/dgn/WA/MakeBlockQA/MakeBlock/nnARMCore/lef2ic/lef2ic.cmd: writing file ... (MBLK-6)
Info: Executing command: "icfb -nograph -replay lef2ic.cmd -log lef2ic.log >>& /dev/null" ... (MBLK-10)
Info: lef2ic.html: writing to log. (MBLK-8)
Info: lef2ic: Completed ... (MBLK-29)
Info: mb.html: writing to log. (MBLK-8)
Info: mb.restart: removing file. (MBLK-19)
Info: restart.cfg: removing file. (MBLK-19)
Info: makeblock: Completed ... (MBLK-29)
5.6.2 HTML Output

The HTML report is written for the top-level MakeBlock run, as shown in Figure 27. For each of the modules run, an entry is created with the appropriate summary of Warnings and Errors. The output is summarized by the modules run which can follow default data flow as described in Figure 5 or be altered based on users input on the command line when MakeBlock is launched.

![Makeblock Stats]

<table>
<thead>
<tr>
<th>Step</th>
<th>Warning(s)</th>
<th>Error(s)</th>
<th>Timing Error(s)</th>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>mb.html</td>
<td>Sat Sep 17 15:49:48 EDT 2005</td>
<td>Sat Sep 17 16:55:37 EDT 2005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ctgen</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>15:50:05</td>
<td>15:55:31</td>
</tr>
<tr>
<td>ecc</td>
<td>148</td>
<td>0</td>
<td>N/A</td>
<td>15:55:32</td>
<td>15:55:45</td>
</tr>
<tr>
<td>egen</td>
<td>2</td>
<td>0</td>
<td>NA</td>
<td>15:55:45</td>
<td>15:56:17</td>
</tr>
<tr>
<td>fgen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>15:56:18</td>
<td>15:57:11</td>
</tr>
<tr>
<td>gngen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>15:57:13</td>
<td>16:00:51</td>
</tr>
<tr>
<td>sngen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>16:00:54</td>
<td>16:10:10</td>
</tr>
<tr>
<td>rgen</td>
<td>2</td>
<td>0</td>
<td>N/A</td>
<td>16:10:16</td>
<td>16:14:36</td>
</tr>
<tr>
<td>de2ic</td>
<td>211</td>
<td>0</td>
<td>NA</td>
<td>16:14:36</td>
<td>16:16:15</td>
</tr>
<tr>
<td>vlog2ic</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>16:16:16</td>
<td>16:20:42</td>
</tr>
<tr>
<td>pv</td>
<td>670</td>
<td>see pv.html</td>
<td>N/A</td>
<td>16:20:42</td>
<td>16:33:20</td>
</tr>
<tr>
<td>ic2gds</td>
<td>5</td>
<td>0</td>
<td>NA</td>
<td>16:33:20</td>
<td>16:33:39</td>
</tr>
<tr>
<td>abgen</td>
<td>4</td>
<td>0</td>
<td>N/A</td>
<td>16:33:39</td>
<td>16:54:55</td>
</tr>
<tr>
<td>le2ic</td>
<td>11</td>
<td>0</td>
<td>NA</td>
<td>16:54:55</td>
<td>16:55:37</td>
</tr>
</tbody>
</table>

Figure 27: nnARM Core MakeBlock Top-Level Log File

For each of the modules, an additional HTML log file was created in the corresponding run directory, as shown in Figure 28. For each of the modules, additional information was provided with emphasis on the number of Warning and Errors for that particular module run.
Figure 28: nnARM Core Ctgen Report

Each of the module HTML log files link directly into the corresponding report with further details. Figure 29 shows an example of a detailed clock tree report with timing information.
5.7 Design Optimization

Design optimization, as a process, is very time consuming and requires many manual steps. For this reason, once the design timing constraints and the design area/power requirements have been met, further optimization is usually not done. However, if an automated flow, such as MakeBlock, is available, a range of “what if” studies and experiments can be run with very limited overhead, as all re-spins are purely CPU bound. Some additional designer resources are required to simply inspect outputs, to make a determination as to what trade-offs are most suitable for design, and to continue down the optimization path.

For this design optimization test case, we used a previously hardened ARM core design targeted at reducing the area as much as possible, while meeting the original 200MHz design speed.

5.7.1 Design Optimization Setup

The MakeBlock tool was designed with the ability to re-run the same set of modules continuously until a terminating condition occurs, which can be set as a design parameter, or as another terminating condition, such as time elapsed, number of flow passes, etc. However, in this case, since each spin through the design flow is fully independent of the previous or next pass, better implementation is achieved with a simple, external script, which, in turn, will launch a range of separate MakeBlock runs, each with its own independent set of data and design targeted properties. This approach, combined with a mechanism or tool for multiple job distribution
across a server farm, can yield significantly better turnaround times. The external script used to launch the run is shown in Figure 30.

```bash
#!/bin/tcsh

## Launch 6 jobs with attempt at different densities
foreach den (0.70 0.80 0.85 0.90 0.91 0.92 0.93 0.94 0.95)
    set rundir = "run_${den}"
    mkdir $rundir
    cd $rundir
    cp ../all.cfg ${den}.cfg
    cp ../io/ctgen/RegisterFile.const .
    echo "set ::cy::cic::makeblock::fpgen::config(rowUtilization)
        $den" >> ${den}.cfg
    qrun -n -maxjobs 10 –uname sun makeblock -r ../short.replay \
        -config ${den}.cfg -log mb.log &
    cd ..
end

Figure 30: External Wrapper for Launching MakeBlock Runs

In order to get a quick idea of how much area can be reduced while still maintaining timing, nine jobs at different densities (70%, 80%, 85%, 90%, 91%, 92%, 93%, 94% and 95%) were launched with 7 modules each. These modules included floor plan generation (fpgen), placement (qpgen), clock tree insertion (ctgen), special net routing (snrgen), signal routing (siggen), parasitic extraction (rcgen), and static timing analysis (sta). While fpgen and qpgen provide visibility into how much placement can be optimized and how much area can be reduced, other modules are required to obtain accurate timing information. In order to avoid licensing problems, a number of distributed jobs were limited to 10 via – maxjobs option.

5.7.2 Design Optimization Results

Upon completion of all jobs, the inspection results were inspected. The results are shown in Figure 31 through Figure 38. It is important to note that 70% utilization yields one timing error during clock-tree balancing. This timing violation is not real as static timing analysis confirms, however, it is possible to loosen floorplan to the point where it creates more timing problems rather than solving. This is typical on deep sub-micron
technologies where interconnect parasitic effects dominate and allowing more unnecessary space between pins only creates addition timing problems.
Figure 33: Design Optimization Results - 85% Utilization

<table>
<thead>
<tr>
<th>MakeBlock Stats</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Finish</td>
<td></td>
</tr>
<tr>
<td>Sat Sep 17 14:59:41 EDT 2005</td>
<td>Sat Sep 17 15:32:25 EDT 2005</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MakeBlock Summary</th>
<th>85% Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>Warning(s)</td>
</tr>
<tr>
<td>fpgen</td>
<td>0</td>
</tr>
<tr>
<td>eegen</td>
<td>1</td>
</tr>
<tr>
<td>apgen</td>
<td>304</td>
</tr>
<tr>
<td>cugen</td>
<td>143</td>
</tr>
<tr>
<td>sorgen</td>
<td>1</td>
</tr>
<tr>
<td>siggen</td>
<td>4</td>
</tr>
<tr>
<td>regen</td>
<td>3</td>
</tr>
<tr>
<td>sta</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 34: Design Optimization Results - 91% Utilization

<table>
<thead>
<tr>
<th>Makeblock Stats</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Finish</td>
<td></td>
</tr>
<tr>
<td>Sat Sep 17 14:59:40 EDT 2005</td>
<td>Sat Sep 17 15:31:19 EDT 2005</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MakeBlock Summary</th>
<th>91% Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>Warning(s)</td>
</tr>
<tr>
<td>fpgen</td>
<td>0</td>
</tr>
<tr>
<td>eegen</td>
<td>1</td>
</tr>
<tr>
<td>apgen</td>
<td>304</td>
</tr>
<tr>
<td>cugen</td>
<td>139</td>
</tr>
<tr>
<td>sorgen</td>
<td>1</td>
</tr>
<tr>
<td>siggen</td>
<td>8</td>
</tr>
<tr>
<td>regen</td>
<td>2</td>
</tr>
<tr>
<td>sta</td>
<td>N/A</td>
</tr>
</tbody>
</table>
MakeBlock Stats

<table>
<thead>
<tr>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sat Sep 17 14:59:41 EDT 2005</td>
<td>Sat Sep 17 15:34:10 EDT 2005</td>
</tr>
</tbody>
</table>

**MakeBlock Summary**

<table>
<thead>
<tr>
<th>Step</th>
<th>Warning(s)</th>
<th>Error(s)</th>
<th>Timing Error(s)</th>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpeng</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>14:59:56</td>
<td>15:01:39</td>
</tr>
<tr>
<td>ecodegen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>15:01:41</td>
<td>15:02:24</td>
</tr>
<tr>
<td>opgen</td>
<td>304</td>
<td>0</td>
<td>NA</td>
<td>15:02:25</td>
<td>15:06:49</td>
</tr>
<tr>
<td>rgen</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>15:06:52</td>
<td>15:15:03</td>
</tr>
<tr>
<td>sorgen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>15:15:05</td>
<td>15:16:52</td>
</tr>
<tr>
<td>siggen</td>
<td>8</td>
<td>0</td>
<td>NA</td>
<td>15:16:54</td>
<td>15:24:25</td>
</tr>
<tr>
<td>rgen</td>
<td>2</td>
<td>0</td>
<td>NA</td>
<td>15:24:31</td>
<td>15:31:33</td>
</tr>
<tr>
<td>sten</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
<td>15:31:33</td>
<td>15:34:10</td>
</tr>
</tbody>
</table>

Figure 35: Design Optimization Results - 92% Utilization

MakeBlock Stats

<table>
<thead>
<tr>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sat Sep 17 15:02:02 EDT 2005</td>
<td>Sat Sep 17 15:35:44 EDT 2005</td>
</tr>
</tbody>
</table>

**MakeBlock Summary**

<table>
<thead>
<tr>
<th>Step</th>
<th>Warning(s)</th>
<th>Error(s)</th>
<th>Timing Error(s)</th>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpeng</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>15:02:21</td>
<td>15:04:22</td>
</tr>
<tr>
<td>ecodegen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>15:04:23</td>
<td>15:05:03</td>
</tr>
<tr>
<td>opgen</td>
<td>304</td>
<td>0</td>
<td>NA</td>
<td>15:05:04</td>
<td>15:08:40</td>
</tr>
<tr>
<td>rgen</td>
<td>137</td>
<td>0</td>
<td>0</td>
<td>15:08:41</td>
<td>15:14:56</td>
</tr>
<tr>
<td>sorgen</td>
<td>1</td>
<td>0</td>
<td>NA</td>
<td>15:14:58</td>
<td>15:16:47</td>
</tr>
<tr>
<td>siggen</td>
<td>8</td>
<td>0</td>
<td>NA</td>
<td>15:16:48</td>
<td>15:26:23</td>
</tr>
<tr>
<td>rgen</td>
<td>2</td>
<td>0</td>
<td>NA</td>
<td>15:26:27</td>
<td>15:33:14</td>
</tr>
<tr>
<td>sten</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
<td>15:33:14</td>
<td>15:35:44</td>
</tr>
</tbody>
</table>

Figure 36: Design Optimization Results - 93% Utilization
As can be seen from Figure 31 through Figure 38, this design can be optimized to a 93% utilization, while still meeting the timing. Utilization of 94% or higher yielded incomplete MakeBlock runs, thus identifying unachievable constraints.

To closer inspect failure points we can focus on the 94% utilization. The physical placement of the design was successful, as seen in Figure 39.
The actual placement percentage was 99.75%, which is extremely high, thus leaving very limited space for timing closure on critical signals.

Figure 39: Physical Placement Summary at 94% Utilization

Due to a very limited, available, silicon area, and due to the need for a larger area, clock tree insertion and balancing was not completed, as shown in Figure 40.

Figure 40: Clock Tree Insertion Failure Summary

The 94% utilization test case was incomplete, due to inability to meet timing requirements on the clock tree; however, the 95% utilization test case was not completed due to lack of spacing during the physical placement, as shown in Figure 41 and Figure 42.

Figure 41: Failed Physical Placement Summary at 95% Utilization
5.7.3 Optimized Design Results

As can be seen from the Design Optimization Results, the optimal case is achieved at 93% utilization, while the original timing is still met at 200MHz. The final placed layout is shown in Figure 43. As we can see, the final utilized area is approximately 590u by 1980u, resulting in area of 3.14mm$^2$. Compared to the typical design of 70% utilization [25,26], dimensions of 1800u by 2260u and area of 4.07mm$^2$, this represents more than a 20% area utilization improvement. Figure 43 and Figure 44 show optimized design standard cell placement and a routed clock tree respectively.
Figure 43: Optimized Design Placed Layout
Figure 44: Optimized Design Clock Tree Layout
In addition to optimizing area, this approach resulted in significantly shorter wiring of the signals. As can be seen from Figure 46 and Figure
47, the total wire routing length has been reduced by more than 146,000um.

### Siggen Log Files

<table>
<thead>
<tr>
<th>File</th>
<th>Warning(s)</th>
<th>Error(s)</th>
<th>Size(KB)</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>siggen.log</td>
<td>9</td>
<td>0</td>
<td>29</td>
<td>Set Sep 17 15:23:08 EDT 2005</td>
</tr>
</tbody>
</table>

### Siggen Rpt Files

<table>
<thead>
<tr>
<th>File</th>
<th>met1(u)</th>
<th>met2(u)</th>
<th>met3(u)</th>
<th>Total(u)</th>
<th>Size(Kb)</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBRARY wires</td>
<td>1001574.05</td>
<td>1305017.45</td>
<td>735182.40</td>
<td>1343</td>
<td>2758388.30</td>
<td>Set Sep 17 15:22:54 EDT 2005</td>
</tr>
</tbody>
</table>

Figure 46: Signal Routing Report for 70% Utilization

### Siggen Log Files

<table>
<thead>
<tr>
<th>File</th>
<th>Warning(s)</th>
<th>Error(s)</th>
<th>Size(KB)</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>siggen.log</td>
<td>8</td>
<td>0</td>
<td>33</td>
<td>Set Sep 17 15:25:23 EDT 2005</td>
</tr>
</tbody>
</table>

### Siggen Rpt Files

<table>
<thead>
<tr>
<th>File</th>
<th>met1(u)</th>
<th>met2(u)</th>
<th>met3(u)</th>
<th>Total(u)</th>
<th>Size(Kb)</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBRARY wires</td>
<td>895027.85</td>
<td>1102035.80</td>
<td>829027.20</td>
<td>2611664.65</td>
<td>1372</td>
<td>Set Sep 17 15:26:08 EDT 2005</td>
</tr>
</tbody>
</table>

Figure 47: Signal Routing Report for 93% Utilization
CHAPTER 6. CONCLUSIONS

As can be seen from the EXPERIMENTAL TESTING AND VALIDATION chapter, the resulting solution is a well integrated, modular and streamlined tool enabling quick turnaround times through the block hardening steps of an RTL2GDS flow. It has been demonstrated that via MakeBlock, the following can be achieved:

6.1 Reduced Design Cycle Time

Under normal circumstances, manual block hardening of the ARM Core requires 2 man-days of work on the first pass and 1 man-day of work on subsequent passes based on our manual test, however, as can be seen from Figure 27, the entire block hardening process via MakeBlock takes under an hour of real time. Initial setup for the MakeBlock run took 0.5 man-days and optimization setup took additional 0.25 man-days. Since the MakeBlock implementation is purely CPU bound, additional speedups can be achieved by using newer software and hardware, such as Linux based operating systems and X86 processor based hardware.

6.2 Improved Design Parameters and Performance

In addition to cycle-time speedup via automation, significant design optimization can be achieved through an easy-to-run set of trial-and-error optimization runs. As shown in 5.7.3 Optimized Design Results, large area reductions of 20+%, can be achieved through this approach as well as a reduction in total wire routing. Other design parameters, such as timing/speed, power, etc., can also be optimized in a similar manner but tradeoff between silicon area, performance and power must be made as described in EXPERIMENTAL TESTING AND VALIDATION, page 63.
A reduction in cycle time, as well as improved design parameters and performance, translates into a faster speed to market and a greater revenue.


[20] *http://www.swig.org*

[21] *http://www.si2.org*

[22] *http://www.opencores.org*

[23] *http://www.arm.com*


[26] *http://www.ee.vt.edu/~ha/cadtools/PnR.htm*
VITA

Dragomir Nikolic was born May 22nd, 1974 in Belgrade, Yugoslavia. He attended Electro-Technical High School Nada Dimic 1989-1992 in Zemun, Yugoslavia and obtained high school degree in fall of 1993. As an exchange student he also obtained his high school diploma from Hot Springs High School in the spring of 1993 at Hot Springs, South Dakota. He attended the South Dakota School of Mines and Technology, Electrical Engineering program in Rapid City, South Dakota, 1993-1995, after which he transferred to the University of Kentucky, Electrical Engineering program in the fall of 1995. He obtained BS degree in Electrical Engineering from University of Kentucky in 1997. He has been employed by Cypress Semiconductor as CAD Engineer since June 1997.

________________________

________________________