

DIGITAL DESIGN FLOW OPTIONS

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of The Ohio State University

By

Sagar Vidya Reddy, B.E.

* * * * *

The Ohio State University

2001

Master's Examination Committee:
Professor. Joanne E. DeGroat, Adviser
Professor. Steven B. Bibyk

Approved by

Adviser

Department of Electrical Engineering

ABSTRACT

VLSI (Very Large Scale Integration) IC design flow is a term used to describe the process of chip design. The circuit designer/design group uses a multitude of computer aided design tools throughout the design flow to tape out an integrated circuit IC. There are numerous computer aided design (CAD) tools available commercially to help such designers in design. The design tools and the order in which they are used is termed as the design flow.

CAD tools however do have certain problems associated with them. Cost, accessibility, compatibility and reliability are a few of such problems. At the university level, usability, tutorials and adequate documentation is needed to facilitate students to adapt and familiarize themselves with the CAD tool suites in a short time.

In mixed signal chips, there are two distinct phases of design, the analog and the digital part. The analog part of the chip (probably containing a few hundred

transistors) consumes a considerable amount of time to design. The digital part of the chip (probably containing tens of thousands to a million transistors) needs to be developed in a relatively short period of time. Thus design team needs to place a few thousand gates quickly and reliably. To support this, the choice of CAD tools becomes critical. A good CAD tool can drastically reduce the time required to design a large digital block.

The scope of this thesis is to search for an optimal digital design flow. The aim is to use a set of commercial and open sourced tools and bring digital logic design to such a stage that the design can be shipped for fabrication or attached to an analog entity for fabricated into a mixed signal chip. The stress in this thesis is going to be on evolving a streamlined design flow, which is quick, cheap, reliable and acceptable. Conclusions will be reached as to how a particular tool is best suited for each stage of design. These design suites will be compared and contrasted with respect to various qualities like cost, accessibility, usability, etc.

Most non-commercial CAD tools fail to meet the industry's requirements when the design becomes technology specific. Most industrially accepted CAD tools are expensive. It is here that a cheaper design alternative is required. Solutions to this problem using Electric and TestBencher are suggested in this thesis.

In this document, the primary focus is on three tools, TestBencher, X-HDL3 and Electric. These CAD tools were chosen, as they seemed ideal for a university environment.

All the chapters start with a brief explanation of a design stage. After a brief explanation, every chapter is further subdivided into various sections. Each section deals with a particular aspect of design and how the design can be completed using one of the CAD tools. Throughout this document an example based approach is adopted. The examples chosen are simple circuits to increase readability.

To Nipa

ACKNOWLEDGMENTS

I would like to thank my friend Nipa for her continuous motivation, moral support, encouragement and love. I would like to thank Prof. DeGroat for her suggestions, support, guidance and the inspiration I got while working with her during the course of my masters degree. I am especially indebted to Prof. Bibyk for all his invaluable suggestions, advice, support, guidance and help.

VITA

Dec 24, 1977

Born – Guntur, India.

1995 – 1999

B.E. Electrical & Electronics
Engineering,
Bangalore University,
Bangalore, India.

1999- present

Masters student,
The Ohio State University

FIELDS OF STUDY

Major Field: Electrical Engineering

TABLE OF CONTENTS

| | <u>Page</u> |
|---------------------------------|-------------|
| Abstract..... | ii |
| Dedication..... | v |
| Acknowledgments..... | vi |
| Vita..... | vii |
| List of Figures..... | xii |
| List of Tables..... | xvii |
| Chapters | |
| 1. Introduction..... | 1 |
| 1.1 Generic IC design Flow..... | 1 |
| 1.2 Problem description..... | 7 |
| 1.3 Ideal solution..... | 8 |
| 1.4 Proposed solution..... | 9 |
| 1.5 Research objective..... | 10 |
| 2. Design idea | 12 |
| 2.1 Evolving a design Idea..... | 14 |

| | |
|----------------------------------------------------------------------------|----|
| 2.2 Capturing a design Idea..... | 14 |
| 2.3 Abstract design idea simulation..... | 17 |
| 2.4 Simulation of theory of operation using C..... | 17 |
| 2.5 HDL for simulation..... | 21 |
| 2.6 Introduction to TestBencher..... | 25 |
| 3. Behavioral description..... | 26 |
| 3.1 Behavioral description to RTL..... | 28 |
| 3.2 Introduction to X-HDL3..... | 29 |
| 3.3 TestBencher for behavioral description..... | 38 |
| 3.4 Test bench generation in WaveFormer Pro..... | 38 |
| 3.5 Verilog to VHDL conversion and vice versa..... | 41 |
| 4. Structural description..... | 43 |
| 4.1 Simulation and timing analysis..... | 44 |
| 4.2 Hierarchy..... | 47 |
| 4.3 Analog simulation of structural description..... | 49 |
| 4.4 Structural description to FPGA..... | 53 |
| 4.5 Structural description to ASIC using Renoir (Mentor Graphics) | 63 |
| 5. IC layout, DRC and LVS | 65 |
| 5.1 Cadence versus other tools..... | 66 |
| 5.2 Why Cadence at the last step?..... | 69 |
| 5.3 Starting a layout for MOSIS..... | 70 |

| | | |
|-------------|-----------------------------------------------------------|-----|
| 5.4 | Selecting a process technology..... | 72 |
| 5.5 | SCMOS versus specific vendor technology..... | 73 |
| 5.6 | Layout formats..... | 76 |
| 5.7 | Setting up Cad tools..... | 77 |
| 5.8 | Submitting to MOSIS..... | 79 |
| 5.9 | Layout using Electric..... | 80 |
| 5.10 | Loading vendor technologies in Electric..... | 81 |
| 5.11 | Layer mapping in Electric..... | 83 |
| 5.12 | Standard cell layout in Electric..... | 85 |
| 5.13 | Manual layout generation in Electric..... | 86 |
| 5.14 | Manual placement and semi-automatic routing options | 87 |
| 5.15 | VHDL synthesis: Silicon Compiler in Electric..... | 91 |
| 5.16 | Simulation..... | 97 |
| 5.17 | Mixed signal design and exporting to Cadence..... | 100 |
| 5.18 | DRC..... | 104 |
| 5.19 | LVS..... | 106 |
| 5.20 | Pad frame..... | 107 |
| 5.21 | Other interesting features of Electric..... | 111 |
| 6. | Summary and Conclusion..... | 117 |
| 6.1 | Summary..... | 117 |
| 6.2 | Conclusion..... | 122 |
| Appendix A. | Behavioral description of SAR..... | 124 |

| | |
|------------------------------------------------|-----|
| Appendix B. Structural description of SAR..... | 132 |
| Bibliography..... | 149 |

LIST OF FIGURES

| <u>Figure</u> | <u>Page</u> |
|------------------------------------------------------------------------------|--------------------|
| 1.1 Generic design flow..... | 2 |
| 2.1 Current status design idea..... | 13 |
| 2.2 Black box block diagram..... | 15 |
| 2.3 Functional block diagram..... | 15 |
| 2.4 Flow chart of SAR..... | 16 |
| 2.5 Simulation result 1 for 0v input..... | 19 |
| 2.6 Simulation result 2 showing the expected result with 1 bit error..... | 20 |
| 2.7 Simulation result 3. | 20 |
| 2.8 TestBencher overview..... | 24 |
| 2.9 D[7:0] shows the result for input voltage of 44..... | 25 |
| 3.1 Current status, behavioral description..... | 27 |
| 3.2 X-HDL3 interface for behavioral to RTL. | 30 |
| 3.3 Pre/ post process interface of X-HDL3..... | 35 |
| 3.4 Illustration of instantaneous test bench creation..... | 40 |
| 3.5 X-HDL3 interface for Verilog ↔ VHDL conversion..... | 41 |

| | |
|---------------------------------------------------------------------------|----|
| 4.1 Current status structural description..... | 44 |
| 4.2 Simulation of structural Dff..... | 46 |
| 4.3 User defined setup and hold times..... | 46 |
| 4.4 Hierarchy of the SAR circuit..... | 48 |
| 4.5 Hierarchy in TestBencher..... | 48 |
| 4.6 Screen capture of Microwind showing direct verilog file input..... | 50 |
| 4.7 Microwind simulation 1..... | 51 |
| 4.8 Microwind simulation 2..... | 52 |
| 4.9 FPGA Synthesis Design flow Using MyCad..... | 53 |
| 4.10 MyCad's FPGA synthesizer..... | 61 |
| 4.11 One bit adder FPGA..... | 62 |
| 4.12 8 Bit adder FPGA..... | 63 |
| 4.13 Renoir ASIC design Flow. | 64 |
| 5.1 Current status layout, DRC, LVS..... | 66 |
| 5.2 Dflipflop synthesized by Microwind..... | 68 |
| 5.3 DRC failure of Microwind's Dff in Cadence..... | 68 |
| 5.4 Design flow for layout generation..... | 70 |
| 5.5 MOSIS recommended design flow..... | 72 |
| 5.6 AMI 0.8 technology 2 x 2 AND OR gate | 75 |
| 5.7 AMI 0.8 technology 2x2 AND OR gate GDS II plot..... | 76 |
| 5.8 Electric interface..... | 78 |

| | |
|----------------------------------------------------------------|-----|
| 5.9 Electric interface for importing LEF files..... | 82 |
| 5.10 All cells of SCMOS_SUBM at lambda=0.30 (AMI 0.80)..... | 83 |
| 5.11 Flexible layer mapping interface in Electric..... | 84 |
| 5.12 Standard cell formation in Electric..... | 85 |
| 5.13 Manual layout generation in Electric..... | 86 |
| 5.14 Designer's floor plan..... | 87 |
| 5.15 Standard cells connected with generic arcs..... | 88 |
| 5.16 Standard cells after maze routing..... | 89 |
| 5.17 River routing of buses..... | 89 |
| 5.18 Result of river routing..... | 90 |
| 5.19 VHDL interface in Electric..... | 93 |
| 5.20 Place and route options..... | 94 |
| 5.21 Layout generated automatically..... | 95 |
| 5.22 Standard cells used. Lines indicate the routed wires..... | 96 |
| 5.23 Inverter layout under simulation..... | 97 |
| 5.24 SPICE simulation..... | 98 |
| 5.25 ALS simulation..... | 99 |
| 5.26 Inverter design imported into Cadence..... | 102 |
| 5.27 8 bit comparator imported into Cadence Virtuoso..... | 103 |
| 5.28 DRC failure due to mismatching Libraries..... | 104 |
| 5.29 DRC customization window..... | 106 |
| 5.30 Automatic Schematic extraction in Electric..... | 107 |

| | |
|-----------------------------------------------------------|-----|
| 5.31 Different Pads available in MOSIS TSMC 0.25..... | 108 |
| 5.32 8 Bit comparator core in Pad Frame..... | 109 |
| 5.33 Zoomed in View of a PAD..... | 110 |
| 5.34 The complete chip..... | 110 |
| 5.35 3-D view of a Dflipflop..... | 111 |
| 5.36 3-Dview of a Dflipflop..... | 112 |
| 5.37 Wasteful layout of an inverter..... | 113 |
| 5.38 Layout after compaction..... | 113 |
| 5.39 Compaction Results..... | 115 |
| 5.40 PLA generated for equations in table 5.11..... | 116 |
| 6.1 Recommended design flow for quick digital design..... | 119 |
| 6.2 Budget oriented design flow..... | 120 |

Appendix A

| | |
|---------------------------------------------------------|-----|
| A1 D[7:0] shows the result for input voltage of 44..... | 128 |
| A2 Result with bus d[7:0] expanded | 129 |
| A3 Result showing the expected 1bit error..... | 130 |
| A4 Result with bus d[7:0] expanded..... | 131 |

Appendix B

| | |
|---------------------------------------------------------------|-----|
| B1 Simulation result of dff | 133 |
| B2 Simulation result of clock divider | 135 |
| B3 Simulation result of clock divider with bus expanded | 136 |
| B4 Simulation result of logic block 1..... | 138 |

| | |
|-----------------------------------------------------|-----|
| B5 Simulation result with bus d[31:0] expanded..... | 139 |
| B6 Simulation result of main register..... | 141 |
| B7 Simulation of digital part of SAR. | 143 |
| B8 Simulation of digital part of SAR. | 144 |
| B9 Simulation result of MYCHIP..... | 146 |
| B10 Simulation result 2 of MYCHIP..... | 147 |
| B11 Simulation result 3 of MYCHIP..... | 148 |

LIST OF TABLES

| <u>Table</u> | <u>Page</u> |
|-----------------------------------------------------------------|-------------|
| 1.1 Properties of an ideal tool kit..... | 6 |
| 1.2 Aim of thesis..... | 8 |
| 2.1 SAR specs sheet..... | 12 |
| 2.2 C code for design idea simulation..... | 15 |
| 2.3 Verilog Code for design idea simulation..... | 19 |
| 3.1 Behavioral description of dff..... | 24 |
| 3.1 RTL code generated X-HDL3..... | 26 |
| 3.3 Code not translatable by X-HDL3..... | 27 |
| 3.4 Erroneous code generated by x_HDL3..... | 28 |
| 3.5 Error replaced by Perl script..... | 31 |
| 3.6 Structural description of Dff..... | 33 |
| 4.1 Structural description of Dff..... | 33 |
| 4.2 8 bit adder structural description..... | 47 |
| 5.1 Resources provided by MOSIS for designers..... | 61 |
| 5.2 Features to be provided by vendors for their libraries..... | 63 |

| | |
|------------------------------------------------------------|-----|
| 5.3 Verification before starting a layout..... | 68 |
| 5.4 Electric salient features..... | 69 |
| 5.5 Semiautomatic routing options in Electric..... | 75 |
| 5.6 Problems associated with semiautomatic routing..... | 78 |
| 5.7 Flat VHDL code for Silicon Compiler..... | 80 |
| 5.8 Silicon Compiler options..... | 84 |
| 5.9 List of simulation export formats in Electric..... | 88 |
| 5.10 DRC options..... | 93 |
| 5.11 PLA generation equations..... | 103 |
| 6.1 Salient features of TestBencher, X-HDL3, Electric..... | 105 |
| 6.2 CAD tools encountered..... | 121 |
| A1 Behavioral description of SAR in Verilog..... | 124 |
| A2 VHDL file generated by X-HDL3..... | 125 |
| B1 Verilog description of dff | 132 |
| B2 Verilog description of clock divider..... | 134 |
| B3 Verilog description of logic block 1..... | 137 |
| B4 Verilog description of main register..... | 140 |
| B5 Verilog description of digital part of SAR. | 142 |
| B6 Simulation of digital part of SAR..... | 143 |
| B7 Simulation of digital part of SAR. | 144 |
| B8 Simulation result of MYCHIP..... | 145 |

CHAPTER 1

INTRODUCTION

There are several hundred CAD tool suites in the market today, which can help us in integrated circuit (IC) design. Based on the requirements, a design team needs to appropriately choose the right set of CAD tools for development. This document is aimed at helping university based design teams in establishing a quick and a reliable design flow. The primary CAD tool suites used are TestBencher, X-HDL3 and Electric. Section 1.1 reviews a generic IC design flow.

1.1 GENERIC IC DESIGN FLOW

Design Flow is a term used to describe the various design phases of an IC design. The first thing needed to start a design, is a specification of what needs to

be done ('specs'). Given a specification, the most general approach adopted is shown in figure 1.1.

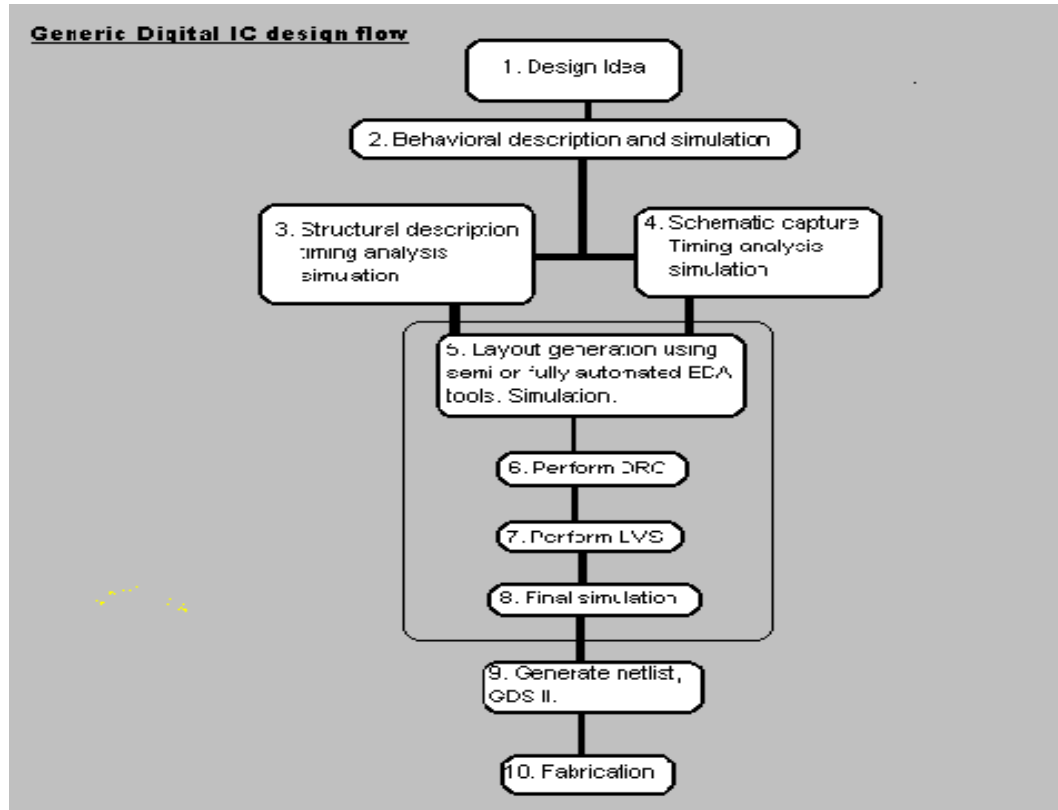


Figure 1.1: Generic IC design flow

Design idea:

Based on the specification given, the design team forms a general idea about the solution to the problem. System level decisions are made regarding the design and a general consensus is reached regarding the major functional blocks

that that go into the making of the chip. At the end of this stage, a general block diagram solution of the design is agreed upon. CAD tools are generally not needed at this stage.

Behavioral description

Hardware Description Languages (HDLs) are used to model the design idea (block diagram). Circuit details and electrical components are not specified. Instead, the behavior of each block at the highest level of abstraction is modeled. Simulations are then run to see if the blocks do indeed function as expected and the whole system performs as a whole. Behavioral descriptions are important as they corroborate the integrity of the design idea.

Structural description

HDLs are used to describe the construction of the design at the next level of abstraction. In this stage, various electronic components and circuit details that go into the making of the various blocks of the system are modeled. These may include primitives like logic gates and very large pre-designed blocks called standard cells. Simulations are then run on each block separately to test its operation. Once this is completed, all blocks are interconnected and a system level

simulation is run. A successful simulation ensures that an electronic circuit can indeed be built to match the behavioral description.

Schematic description

A schematic description is similar to a structural description. In a rigorous context, schematics can be considered to be at a lower level of abstraction, as the topology of the circuits needs to be considered. In most cases, graphical tools are used to build the circuits in a virtual breadboard like environment. Just as in the previous stage, various electronic components like logic gates and standard cells are placed and interconnected to make the circuit blocks. This is an alternative form of structural description with more attention being paid to topology. A successful simulation ensures that an electronic circuit that matches the behavioral description can be built.

Layout description

This is the lowest level of abstraction possible in circuit description. In this stage, the circuit is described with respect to the silicon and other materials that go into the making of the IC. Extreme attention is paid to geometry of the components.

In other words, this is how the IC would look, if we took a closer look at it through the microscope. Parasitic elements need to be extracted at each stage. The placement and interconnects of each and every individual entity of the previous stage need to be specified. Extensive simulations of each block are run in this stage. A floor plan of the chip is made and the blocks are routed together. After the routing phase, simulations of the entire design are run.

LVS

After the completion of the layout description of each block, a Layout vs. Schematic (LVS) check is performed. This ensures that the layout is in conformance with the schematic. The design process moves back and forth between Layout, LVS and DRC.

DRC

This stage is often dependent on the final process technology that is used to manufacture the chip. The design rule check ensures that the rules laid down by the fabrication process technology are not violated. A good example would be, some

processes need transistors, wires and polysilicon to be of a certain minimum width. The layout would have to be drawn based on such constraints. The design toggles between Layout, LVS and DRC.

CIF, GDS II

This is the final stage before fabrication. Most fabrication plants accept submissions in CIF or GDS II format (refer to chapter5). These are computer-generated files that describe the IC layout. They incorporate all the necessary details for manufacturing the chip. MOSIS (section 5.3) accepts submissions in both CIF and GDS II formats and these files can be submitted electronically.

A chip design team's job comes to a momentary pause until the first few prototyped chips arrive from the fabrication plant. After the 'silicon' arrives, the testing phase begins. The first basic tests include a functional verification of the chip under ideal conditions. Once the functionality is verified, extensive tests are performed by stretching various electrical parameters like frequency of operation and temperature to extremities. The chip's behavior is then documented into the chip's user manual.

1.2 PROBLEM DESCRIPTION

Every stage of the design flow (section 1.1) involves a CAD tool. Thus, the choice of the CAD tools dictates the ease and expediency of the design process. Every tool has some salient features that make it advantageous for a certain design phase. CAD tools should facilitate chip design in as short a period of time as possible. At the same time the design needs to be reliable and robust. Depending on the choice of the tool, and by taking advantage of its salient features, an optimal solution to traverse through the digital flow can be reached.

Table 1.1 enumerates the essential properties that CAD tool suites and need to possess.

Properties of CAD tool suites.

- 1. Cost Effective.*
- 2. Accessible.*
- 3. Conform to industry standards.*
- 4. Acceptable by the industry.*
- 5. Accurate.*
- 6. Scalable to the latest technology.*
- 7. Adequate support and documentation.*
- 8. Platform independent.*
- 9. Networking capability.*
- 10. User-friendly.*
- 11. Interface with other tools.*

Table 1.1: Essential properties of a CAD tool suites.

The problem at hand can now be summed up as: *What collection of tools, possessing as many as the above-mentioned properties, would help us evolve a better design flow?*

1.3 IDEAL SOLUTION.

An ideal solution to the above problem would be to use one tool from a single vendor, which possesses all of the above properties (table 1.1). Unfortunately such a tool is not available. The reason being that every tool has some unique quality, which makes it more useful than others in some aspect of the design flow. It is thus left to the discretion of the design team to choose the tools that best fit their needs and budget. In some cases, especially in big companies like HP, IBM, SGI, Intel, the design flow is so customized, that they develop ‘in-house’ CAD tool suites to suit their needs. In this document, however, the thrust is going to be from the point of view of a generic budget oriented university based design team. Apart from the properties mentioned in table 1.1, an ideal solution for such a team would include a collection of tools that are cheaper than most commercial tools and are easily accessible with good documentation. It would be even better if the software were Open Sourced.

1.4 PROPOSED SOLUTION

It is not possible to explore all of the available tools. Thus, a few of the tools that fall within the reach of a university based design team are explored. Most universities already have an established design flow with tools from vendors like Mentor Graphics and Cadence. However, in this document, various other tools that are relatively easier to work with, cheaper to obtain, and accepted by the industry are explored. The aim of this document is summarized in table 1.2.

After careful scrutiny of some of the well-known CAD tool suites (table 6.2), TestBencher, XHDL and Electric were chosen for implementing a complete design flow. The Proposed Solution is to use TestBencher for Verilog descriptions, X-HDL3 for verilog to VHDL conversion and Electric for schematic and Layout descriptions. An example-based approach is used in this document to illustrate every stage of the design. The SAR circuit is used as an example circuit to show the working of TestBencher and an 8-bit comparator circuit is used for Electric.

Aim of this thesis

1. *Establish a design flow for an efficient tape out of chips.*
2. *Provide a digital flow for mixed signal design.*
3. *Establish a cost effective design flow.*
4. *Establish a time saving design flow.*
5. *Gain depth into the usefulness of synthesis, place and route tools.*
6. *Establish a design flow on platforms like Windows and Linux.*
7. *Serve as a guide/ tutorial for students in digital VLSI design.*

Table 1.2: aim of thesis

This document contains 6 chapters. The chapters are a walkthrough of the design flow with illustrations of the usage of TestBencher and Electric at every stage. Deviations into an ASIC and an FPGA are briefly discussed in chapters 3 and 4.

1.5 Research objective

Before delving into the design process, a few things need to be kept in mind. Building a chip alone does not solve the entire problem. For example, consider the SAR type A/D. A chip can be taped out using the design flow suggested in this document. However, a successful chip tape out is just a part of the solution. The entire solution should include a complete interface, to say, a computer or a DSP board. Thus, a PCB needs to be designed as well.

In big corporations, there are different teams involved in bringing out a single product. It could consist three teams, one for analog design, one for digital design and another team for system integration. The system integration team would be responsible for using custom ICs other ‘off the shelf’ ICs to bring out the final product. Truly speaking, an ‘end to end design flow’ should include the entire process, right up to the point of shipping the complete usable product to the customer. It is thus left to the design managers to weigh the tradeoffs involved at each stage. For example, consider the aspect of using completely off the shelf chips

to tape out the A/D converter as opposed to using a custom ICs or, using a custom built digital chip along with an analog off shelf comparator and an 8 bit D/A to make the SAR A/D. There are many such options that require managerial decisions. But the important points to be kept in mind while making these decisions are listed below.

?? Time span of the design.

?? Cost.

?? Meeting the specifications.

One of the research goals of this thesis in design flow discussed above is to provide a streamlined path to tape out mixed signal chips that interface to standard MCU and DSP chips and their development platforms. Thus, the one of the objectives, in a broader sense, is to enable the development of programmable mixed signal systems.

The chapters in this document are sectioned according to the general design flow. Chapter 1 reviews a generic design flow, the most common methods of approach, and the intent of this document. Chapter 2 deals with the first stage of design, which is capturing an idea and the initial design entry. Chapter 3 deals with the behavioral descriptions of circuits. Chapter 4 discusses structural descriptions and chapter 5 discusses the layout process. The 6th chapter, namely, the ‘summary and conclusion’, summarizes this document and gives a broad overview of what future work can be done.

CHAPTER 2

DESIGN IDEA

This is probably the only stage in the design flow that does not find the necessity of a computer. This is also the first phase of design after receiving the specifications for the IC. In this stage, the design team makes decisions regarding the general build up of the chip. Experience is recommended at this stage of design. In fact, in large corporations, team managers are usually the key players at such a stage. They are the people who usually make decisions about the main functional blocks of the chip. At the end of this stage, a consensus of the block level specification of the chip is evolved. Figure 2.1 shows the current status of the design flow up to chapter 2.

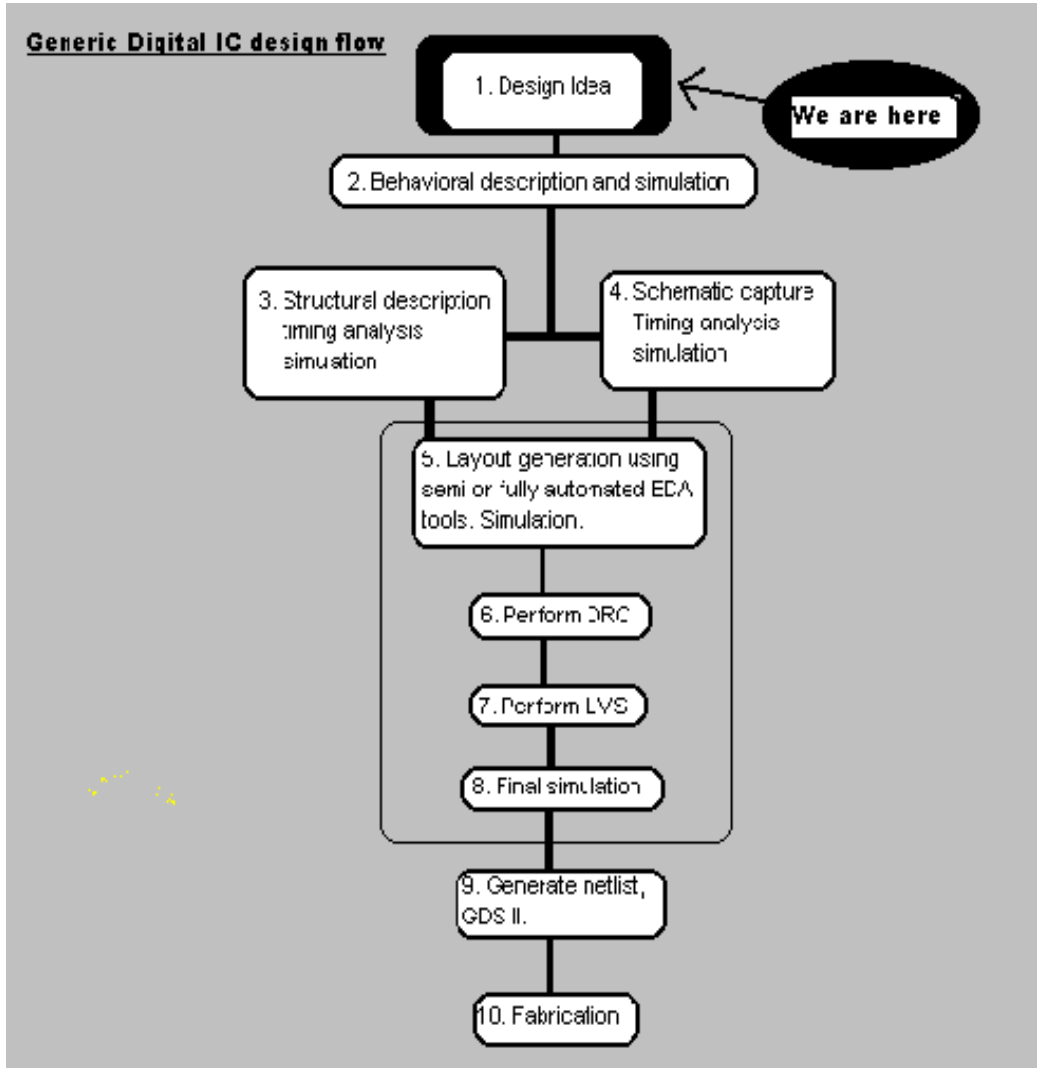


Figure 2.1: Current status, Design Idea

2.1 EVOLVING A DESIGN IDEA

As mentioned earlier, an example based explanation approach is used in this thesis. The example chosen is that of a successive approximation register (SAR). Assume that the design team is a university based team and the following ‘specs sheet’ (table 2.1) was given.

Specs Sheet

1. *Build a SAR type digital circuit, which interfaces to an analog block.*
2. *Required resolution of SAR is 8 bits.*
3. *Conversion must start when the signal ‘Soc’ is asserted.*
4. *The circuit resets itself when the signal ‘Rst’ is asserted.*
5. *‘Eoc’ must be asserted when the result is valid on the output bus.*
6. *The clock frequency is 1Mhz.*

Table 2.1: SAR specs sheet

2.2 CAPTURING A DESIGN IDEA

Interface block diagram: The first requirement is that of a ‘black box’ like block diagram, which illustrates the interface to the outside world. Figure 2.2 illustrates that. The hypothetical chip that is to be built is called MYCHIP.

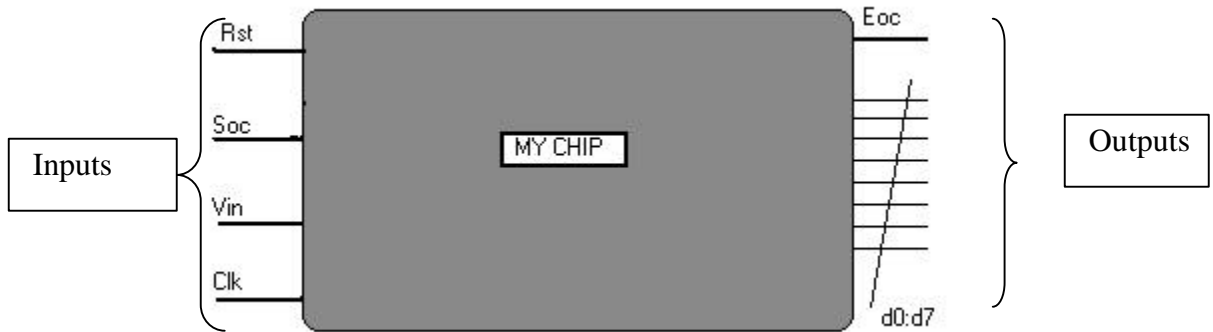


Figure 2.2: Black box like block diagram

Functional block diagram: A more detailed block diagram is drawn which shows the major functional blocks of the IC.

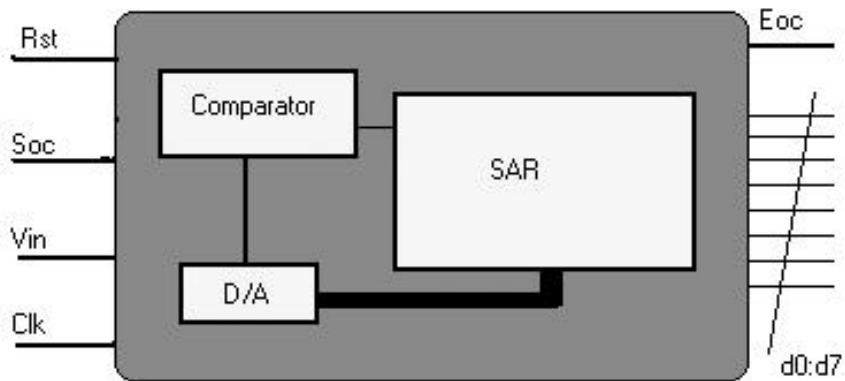


Figure 2.3: Functional block diagram

Ensuring the functionality: In most cases, a flow chart is drawn to ensure that the understanding of the working of the circuit (in this case, the SAR) is correct. Figure 2.4 shows the flow chart of the SAR

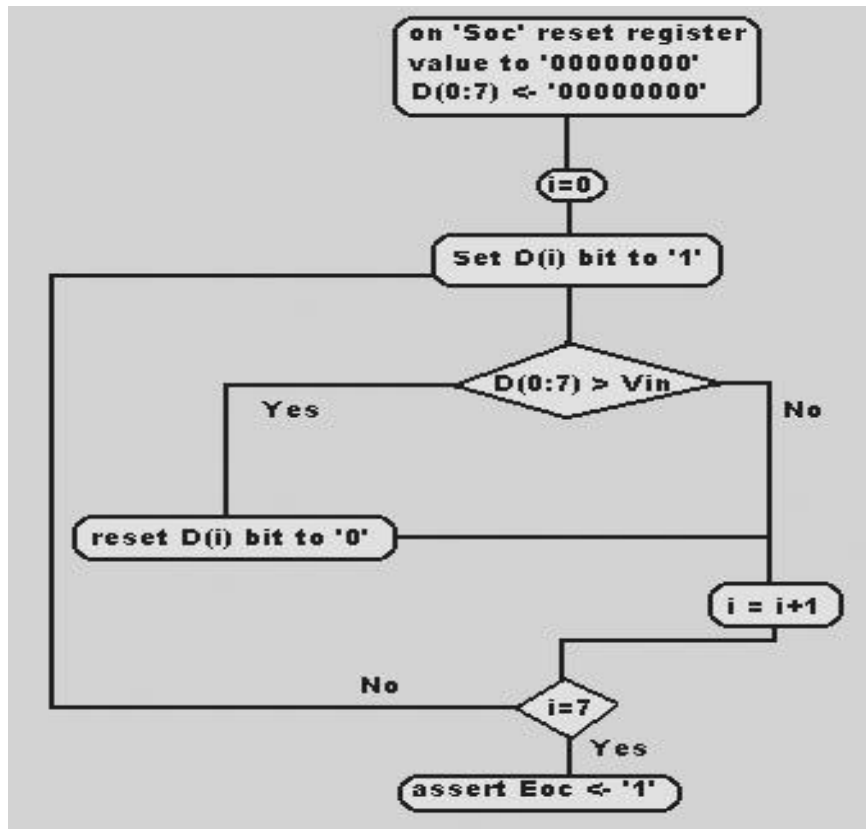


Figure 2.4: Flow chart of SAR

2.3 ABSTRACT DESIGN IDEA SIMULATION

The flow chart in figure 2.4 needs to be simulated to ensure that the theory of operation is correct. The only way to test it is to program it into some sort of CAD tool. The most common answer would be a programming language. But because of the issue of concurrency, a HDL is used for simulation instead of a programming language. Section 2.4 describes a design idea simulation using a programming language like C.

2.4 SIMULATION OF THEORY OF OPERATION USING C

A sample of the code used for simulation is shown in table 2.2. The results obtained are as expected (Note that depending on its construction, the probability of 1 lsb error in a SAR is 0.5). Figures 2.4.1,2.4.2,2.4.3 show the simulation results. For simplicity it is assumed that the input voltage to MYCHIP ranges from 0 to 255.

```

#include <stdio.h>
#include <math.h>
#include <string.h>
int d[7];
int vin=255;
void main (void)
{
    void initialize (void);
    void incr (int);
    void decr (int);
    int compare ( void );
    int i,k;
        initialize();
        for (i=7;i>0;i=i-1)
        {

                incr(i);
                k=compare();
                if (k==1)
                    decr(i);

                printf("d[7:0]=%d%d%d%d%d%d%d\n",d[7],d[6],d[5],d[4],d[3],d[2],
d[1],d[0]);
        }
        printf ("\nVin= %d\n",vin);
        printf ("\nFinalanswer
d[7:0]=%d%d%d%d%d%d%d\n",d[7],d[6],d[5],d[4],d[3],d[2],d[1],d[0]);
    }
void initialize (void)
{
    int i;
    for (i=0;i<8;i++)
        d[i]=0;
}
void incr (int i)
{
    d[i]=1;
}

```

Table 2.2: C code for design idea simulation. (continued)

Table 2.2: (continued)

```
void decr (int i)
{
    d[i]=0;
}
int compare ( void )
{
    int i;
    int val=0;
    for (i=0;i<8;i++)
    {
        val += d[i]*pow(2,i);
    }
    if (val> vin)
    return (1);
    return (0);
}
```

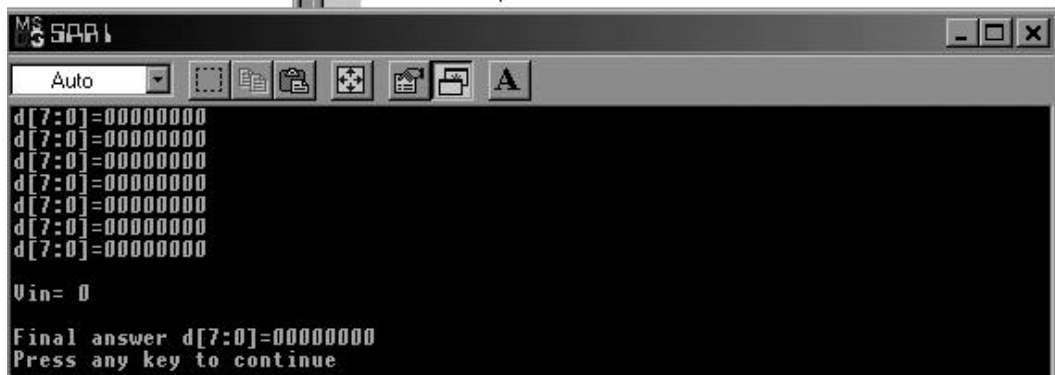


Figure 2.5: Simulation result 1 for 0v input

```
MS-DOS 5.03 SAAL
Auto
d[7:0]=00000000
d[7:0]=01000000
d[7:0]=01000000
d[7:0]=01000000
d[7:0]=01001000
d[7:0]=01001100
d[7:0]=01001110
Vin= 79
Final answer d[7:0]=01001110
Press any key to continue
```

Figure 2.6:Simulation result 2 showing the expected result with 1 bit error.

```
MS-DOS 5.03 SAAL
Auto
d[7:0]=10000000
d[7:0]=10000000
d[7:0]=10000000
d[7:0]=10010000
d[7:0]=10011000
d[7:0]=10011100
d[7:0]=10011100
Vin= 156
Final answer d[7:0]=10011100
Press any key to continue_
```

Figure 2.7:Simulation result 3.

It is easily observed that no provision has been made for the 'Rst' condition. The reason being that it is very difficult to program such 'events' in a normal

programming language. We need constructs that involve the concept of software interrupts like ‘Action Listeners’. This condition, where a close watch is to be kept on signal events, is referred to as ‘concurrency’. It may seem easy to program for one ‘Rst’ condition but the difficulty is easily comprehended if we need to program for a bunch of signals. Since digital circuits are massively parallel dynamic machines, there are bound to be a multitude of signals that need to be programmed. The code would then become highly unwieldy and unreliable.

2.5 HDL FOR SIMULATION

To solve the problem of concurrency mentioned in Section 2.4, HDLs were developed. HDLs are programming languages through which behavior and construction of electronic circuits can be modeled. Two of the most commonly used HDLs today are Verilog and VHDL. HDLs have inbuilt capacity to handle concurrency, but have a syntax similar to normal programming languages. Table 2.3 shows a sample code in Verilog used to simulate the SAR flowchart. The following subsections show simulations carried out on TestBencher

Verilog Code for figure 2.4

```
module sartheory (d,soc,rst);
input soc,rst;
output [7:0]d;
reg [7:0]d;
integer i;
integer vin;
initial
begin
i=7;
vin=123;
end
always @ (posedge rst)
begin
d=7'b0000000;
end
always @ (posedge soc)
begin
while (i > 0)
begin
d[i]=1'b1;
if(d > vin)
begin
d[i]=1'b0;
end
end
i=i-1;
end
end
endmodule
```

Table 2.3: Verilog Code for design idea simulation

2.6 INTRODUCTION TO TESTBENCHER

TestBencher Pro is a design and verification tool suite marketed by SynaptiCad (www.synapticad.com). VeriLogger Pro is a part of this design suite that is capable of compiling Verilog code. VeriLogger Pro is also a complete design and verification environment for ASIC and FPGA designers. TestBencher contains a new type of Verilog simulation environment that combines all the features of a traditional Verilog simulator with an automatic graphical test vector generator. Test vectors can be imported or exported from HP logic analyzers, pattern generators, 3rd party VHDL, Verilog, and SPICE for reuse. Simulation features include waveform viewing, optimized gate-level simulation, single-step debugging, point-and-click breakpoints, hierarchical browser for project management, and batch execution. Figure 2.8 shows a screen shot of TestBencher interface. And figure 2.9 shows a screen capture of the simulation result. For detailed simulation results, refer to appendix A.

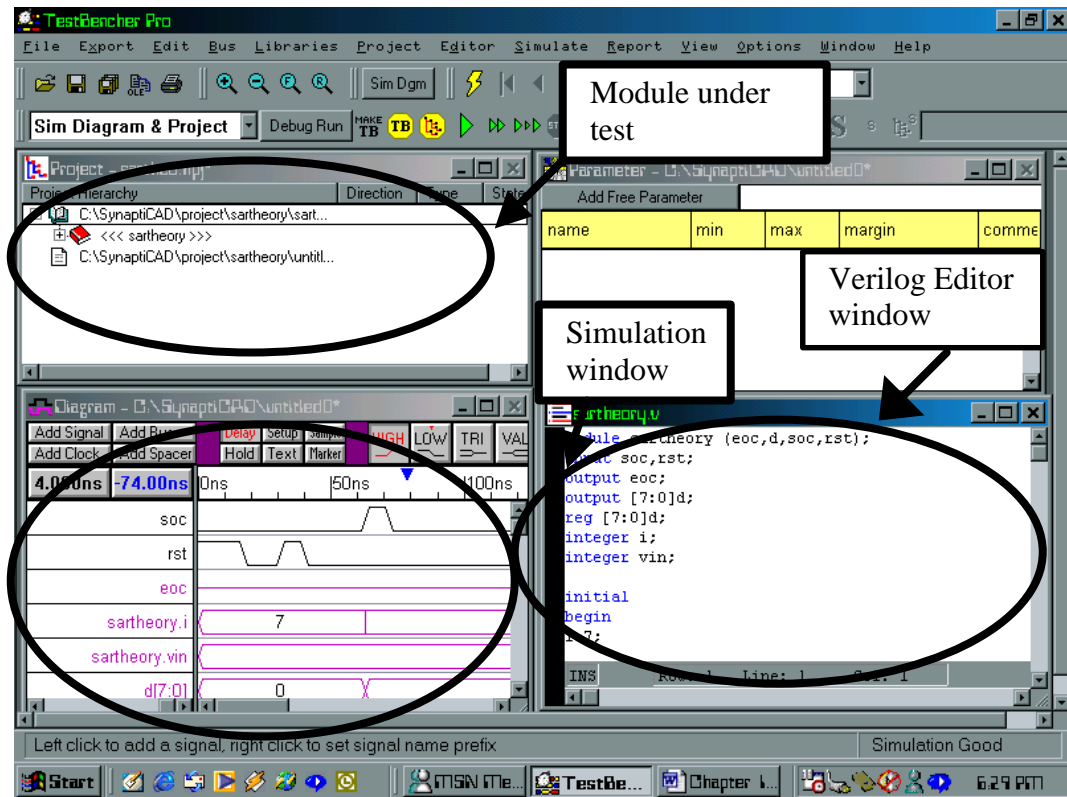


Figure 2.8: TestBench Overview

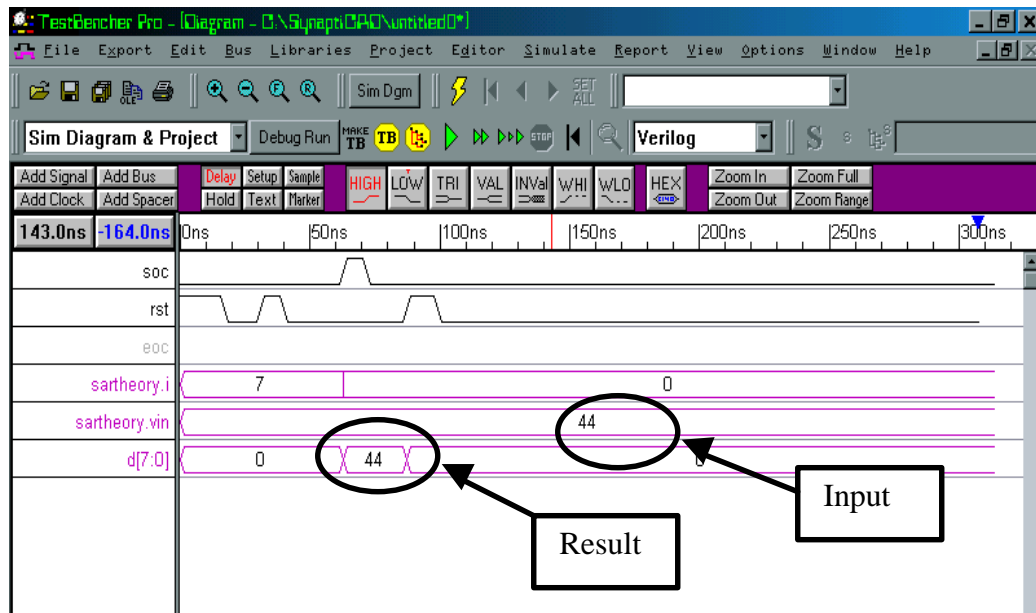


Figure 2.9: d[7:0] shows the result for input voltage of 44.

At the end of the design idea capture stage, the overall system block diagram is decided upon. The next step of the design is to model each of the blocks separately.

CHAPTER 3

BEHAVIORAL DESCRIPTION

The behavioral approach to modeling hardware components is different from circuit design in that it does not necessarily reflect how the design is implemented. It is basically an algorithmic and black box approach to modeling. It accurately models what happens on the inputs and outputs of the black box, but what is inside the box or how the box is constructed is unknown. A behavioral description is usually used in two ways. First, it can be used to model complex components that would be tedious to model using the other methods. This might be the case for example, if you wish to simulate the operation of your custom design connected to a commercial part like a microprocessor. In this case, the microprocessor is complex and its internal operation is irrelevant (only the external behavior is important) so it would probably be modeled using the behavioral style. Second, the behavioral capabilities of HDLs can be more powerful and are more convenient for some designs, especially if an efficient tool that converts some aspects of behavioral code into RTL exists.

Figure 3.1 shows the current status of the design flow up to this chapter.

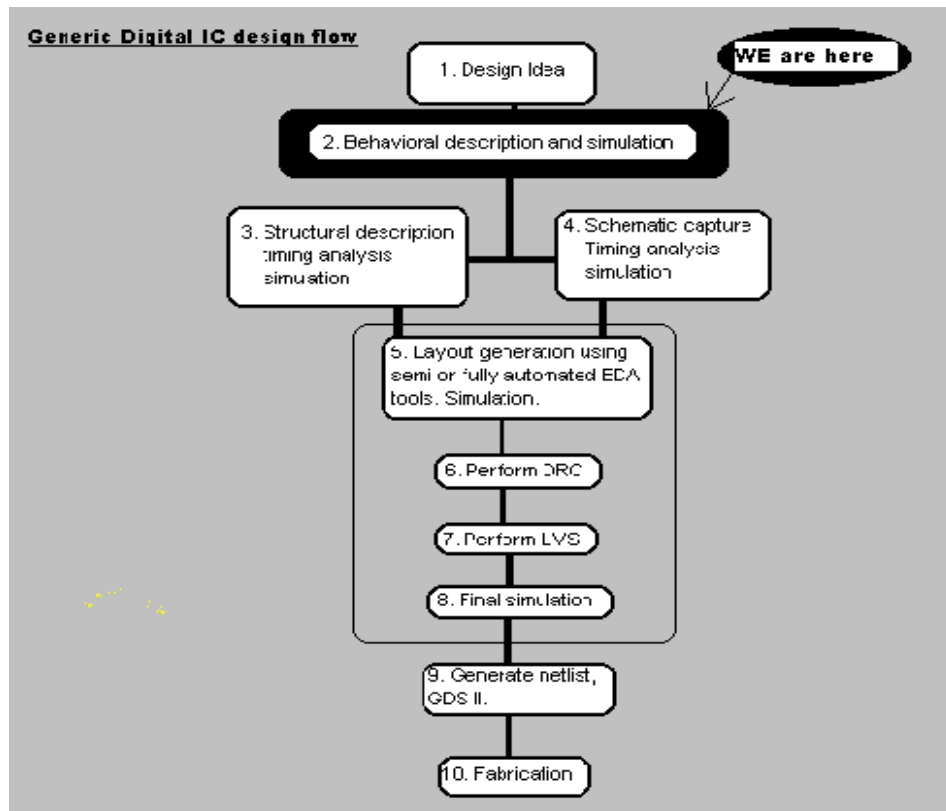


Figure 3.1: current status, behavioral description

3.1 BEHAVIORAL DESCRIPTION TO RTL

The designer starts with an abstract description of the circuit called the behavioral model. This kind of description is used primarily to verify the methodology and functioning of the circuit. The next step is to transform this description into something closer to electronic circuitry. In other words, the behavioral description needs to be converted to RTL (Register Transfer Language). A functional or RTL description describes a circuit in terms of its registers and the combinatorial logic between the registers. This ‘behavioral synthesis’ can either be done manually or automatically by software. The essential goal of doing this is to use logic synthesizers that takes this form of description and synthesizes it to sets of registers and combinatorial logic, which can be readily shipped to FPGA and ASIC vendors. Table 3.1 shows a Verilog behavioral description of a Dflipflop. In sec 3.1, an illustration of converting this behavioral code to RTL using X-HDL3 is shown.

```
module dffbehavioral (q,qbar,d,clk,rst);  
input d,clk,rst;  
output q,qbar;  
reg q,qbar;  
always @ (posedge clk)  
begin  
q=d;  
qbar=~d;
```

Table 3.1: Behavioral description of a Dflipflop

(continued)

Table 3.1: (continued)

```
if (rst == 1'b1)  
begin  
q=1'b0;  
qbar=1'b1;  
end  
end  
endmodule
```

3.2 INTRODUCTION TO X-HDL3

X-hdl3 is a useful tool in converting behavioral code to RTL. Given in table 3.2 is an RTL description of the sample file in table 3.1. Figure 3.2 shows the option in XHDL, which is used to make the code synthesizable (RTL).

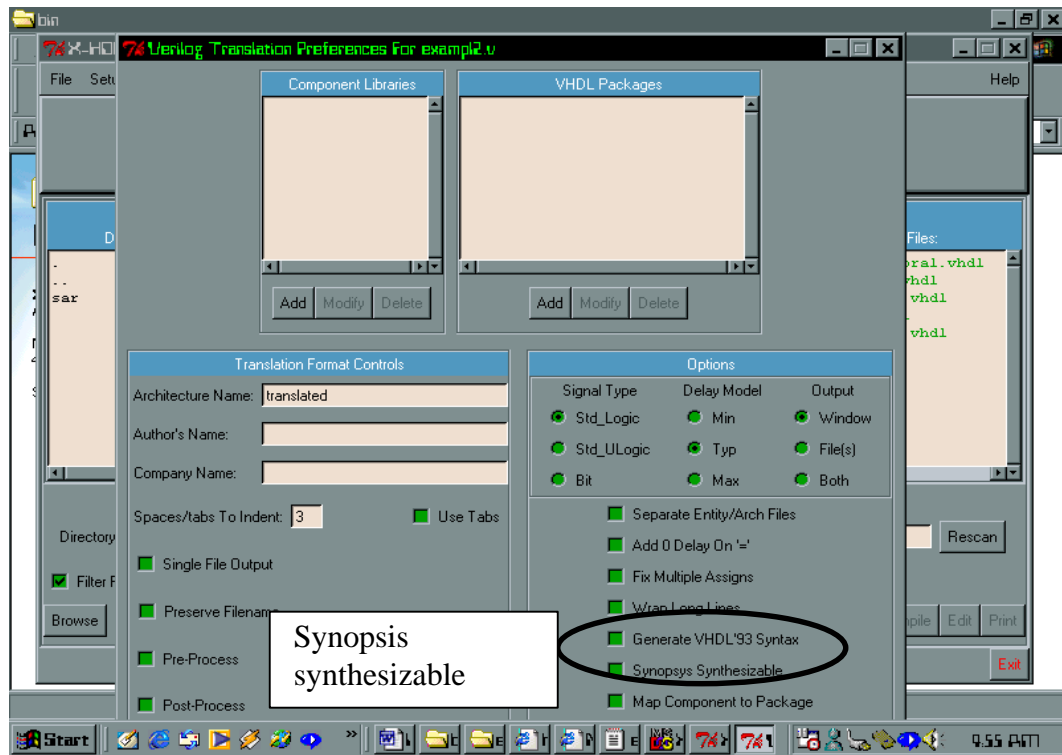


Figure 3.2: X-HDL3 Interface for Behavioral to RTL.

```

//-----
// Wed May 16 02:08:44 2001
//   Input file      : C:/Program Files/xhdl3/bin/dffbehavioral.v
//   Design name     : dffbehavioral
//   Author          :
//   Company         :
//   Description     :
//-----
//
module dffbehavioral (q, qbar, d, clk, rst);
    output q;
    wire q;
    output qbar;
    wire qbar;
    input d;
    input clk;
    input rst;
    reg q_xhdl1;
    reg qbar_xhdl2;
    assign q = q_xhdl1 ;
    assign qbar = qbar_xhdl2 ;
    always @(posedge clk)
    begin : xhdl_0
        reg q_xhdl1_xhdl3;
        reg qbar_xhdl2_xhdl4;
        q_xhdl1_xhdl3 = d;
        qbar_xhdl2_xhdl4 = ~d;
        if (rst)
        begin
            q_xhdl1_xhdl3 = 1'b0;
            qbar_xhdl2_xhdl4 = 1'b1;
        end
        q_xhdl1 <= q_xhdl1_xhdl3 ;
        qbar_xhdl2 <= qbar_xhdl2_xhdl4 ;
    end
endmodule

```

Table 3.2: RTL code generated X-HDL3

However, such tools are not efficient all the time. In some situations, X-HDL 3 may not perform the translation exactly as desired, especially if the code contains language constructs that are not directly translatable. It is to be understood that only a subset of VHDL or Verilog is convertible to synthesizable code. In these situations, the problem can be solved by pre- or post-processing the HDL code. X-HDL 3 facilitates this by filtering the HDL code through pre- or post-processing programs. These filtering programs can be written in any language desired. As an Example, consider a hypothetical circuit described in Table 3.2. Note that there is a ‘wait’ statement in the module, which is quite difficult to convert into an RTL form.

```
module example2 (c,z,clk);  
reg [7:0]a;  
reg [7:0]b;  
input [7:0]z;  
output [15:0]c;  
input clk;  
reg [15:0]c;  
initial  
begin  
c=16'b0;  
end  
always @ (posedge clk)
```

Table 3.3: Code not translatable by X-HDL3 (continued)

Table 3.3: (continued)

```
begin
a = 3+z;
wait (z != 0);
c=a+b*z;
end
endmodule
```

When processed through X-hdl3, it yielded the following error (table 3.3): “ // <<X-HDL>>: Wait expression contains illegal operators. This will require manual modifications”. However, there are options to tackle such errors. It is possible to use Perl scripts to pre/post process the generated code to take care of such errors. Table 3.4 shows the results when such Perl scripts are used in conjunction with X-hdl3.

```
// Verilog file generated by X-HDL - Revision 3.1.28
// Wed May 16 03:49:15 2001
//   Input file      : C:/Program Files/xhdl3/bin/example2.vhdl
//   Design name     : example2
//   Author          : sagar
//   Company         :
//   Description     :
////////////////////////////////////
module example2 (c, z, clk);

    output[15:0] c;
    wire[15:0] c;
    input[7:0] z;
```

Table 3.4: erroneous code generated by X-HDL3 (continued)

Table 3.4: (continued)

```

input clk;
reg[7:0] a;
reg[7:0] b;
reg[15:0] c_xhdl1;
assign c = c_xhdl1 ;
always
begin : xhdl_0
    reg xhdl_initial;
    reg[15:0] c_xhdl1_xhdl2;
    // <<X-HDL>> : Warning - VHDL and Verilog handle variable initialization
differently
    //    The following statement probably requires hand modification.
    xhdl_initial = 1'b1;
    if (xhdl_initial)
    begin
        c_xhdl1_xhdl2 = 16'b0000000000000000;
        c_xhdl1 <= c_xhdl1_xhdl2 ;
        xhdl_initial = 1'b0;
    end
    else
    begin
        forever #100000;
    end
end
always @(posedge clk)
begin : xhdl_1
    reg[7:0] a_xhdl3;
    reg[7:0] b_xhdl4;
    reg[15:0] c_xhdl1_xhdl5;
    a_xhdl3 = 8'b00000011 + z;
    // <<X-HDL>> : Wait expression contains illegal operators. This will require
manual modifications
    @(posedge z != 8'b00000000);
    c_xhdl1_xhdl5 = {8'b00000000, a_xhdl3} + b_xhdl4 * z;
    a <= a_xhdl3 ;
    b <= b_xhdl4 ;
    c_xhdl1 <= c_xhdl1_xhdl5 ;
end
endmodule

```

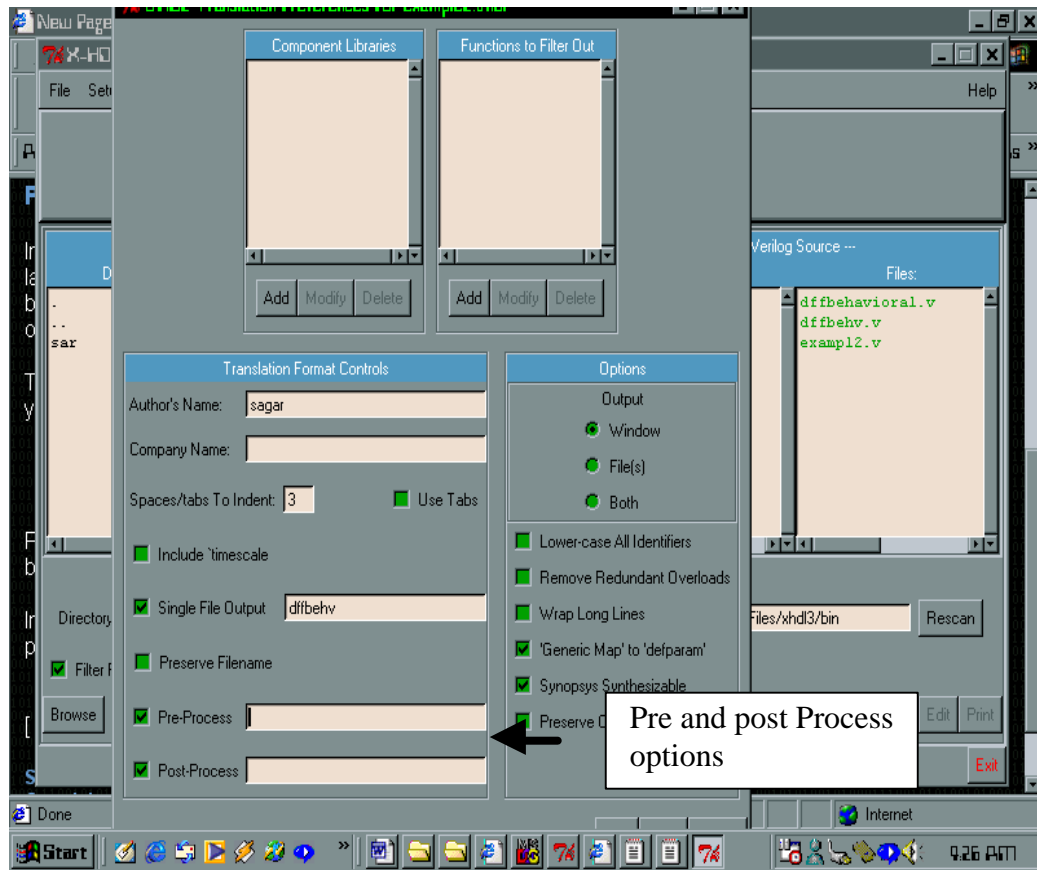


Figure 3.4: Pre/ Post process interface of X-HDL3

Shown below is the code that was modified by a Perl script (table 3.5).

```
/////////////////////////////////////////////////////////////////
//
// Verilog file generated by X-HDL - Revision 3.1.28
// Wed May 16 03:49:15 2001
//
//   Input file      : C:/Program Files/xhdl3/bin/example2.vhdl
//   Design name     : example2
//   Author          : sagar
//   Company         :
//
//   Description     :
//
//
/////////////////////////////////////////////////////////////////
module example2 (c, z, clk);
reg state;
  output[15:0] c;
  wire[15:0] c;
  input[7:0] z;
  input clk;

  reg[7:0] a;
  reg[7:0] b;
  reg[15:0] c_xhdl1;

  assign c = c_xhdl1 ;

  always
  begin : xhdl_0
    reg xhdl_initial;
    reg[15:0] c_xhdl1_xhdl2;
    // <<X-HDL>> : Warning - VHDL and Verilog handle variable initialization
differently
    //      The following statement probably requires hand modification.
    xhdl_initial = 1'b1;
    if (xhdl_initial)
    begin
```

Table 3.5: Error replaced by Perl Script (continued)

Table 3.5: (continued)

```

c_xhdl1_xhdl2 = 16'b0000000000000000;
  c_xhdl1 <= c_xhdl1_xhdl2 ;
  xhdl_initial = 1'b0;
end
else
begin
  forever #100000;
  end
end

always @(posedge clk)
begin : xhdl_1
  reg[7:0] a_xhdl3;
  reg[7:0] b_xhdl4;
  reg[15:0] c_xhdl1_xhdl5;
  a_xhdl3 = 8'b00000011 + z;
case (state)
0 : begin
a = 3+z;
if (z != 0) c=a+b*z;
else state <= 1;
end
1 : if (z != 0)
command3;
endcase

  @(posedge z != 8'b00000000);
  b_xhdl4 = 8'b00000011 - z;
  c_xhdl1_xhdl5 = {8'b00000000, a_xhdl3} + b_xhdl4 * z;
  a <= a_xhdl3 ;
  b <= b_xhdl4 ;
  c_xhdl1 <= c_xhdl1_xhdl5 ;
end
endmodule

```

3.3 TESTBENCHER FOR BEHAVIORAL DESCRIPTION

Using the VeriLogger module in TestBencher it is possible to simulate behavioral descriptions of circuits. The primary advantage of using this software is the instantaneous test bench generation capability. The test benches generated by TestBencher are capable of applying different stimulus vectors depending on simulation response so that the test bench functions as a behavioral model of the environment in which the system is being tested. Surveys of HDL users have indicated that generation of HDL test benches typically consumes 35% of the entire front-end ASIC design cycle. It is clear that automation generation of test benches would significantly reduce the cost of HDL-based design.

3.4 TEST BENCH GENERATION IN WAVEFORMER PRO

Writing a HDL test bench is one of the most tedious and error prone parts of the front-end design process. In HDL test benches, you are required to specify the time for each signal transition. It is easy to make mistakes when writing an HDL test bench, because it is difficult to visualize the temporal relationships between transitions on different waveforms. In addition, test benches are required to incorporate as many boundary conditions as possible. This aspect of test bench modeling is highly error prone, cumbersome and tedious. TestBencher has useful features that facilitate automatic creation of test benches and automatic comparison

of the results. These features are useful for comparing the results of two different forms of descriptions, namely behavioral and structural descriptions to the same test bench. Timing diagram editors such as The WaveFormer allow users to interactively perform timing analysis on their waveforms and document timing relationships using delays, setups, holds, and textual information. These features along with the ability to generate test benches and import results from simulations and test equipment makes TestBencher an ideal environment for waveform manipulation, testing and analysis.

Given below, in table 3.4 is a behavioral description of a Dflipflop.

```
module dffbehavioral (q,qbar,d,clk,rst);  
input d,clk,rst;  
output q,qbar;  
reg q,qbar;  
always @ (posedge clk)  
begin  
  q=d;  
  qbar=~d;  
  if (rst == 1'b1)  
  begin  
    q=1'b0;  
    qbar=1'b1;  
  end  
end  
endmodule
```

Table 3.6: Behavioral description of a Dflipflop

It can be easily observed that writing a test bench for the above flip flop will require a file that is considerably larger than the description itself. Fortunately, we can use the special quality of TestBencher to instantaneously create a test bench that exhaustively covers all the cases. Given below is an output of the simulation.

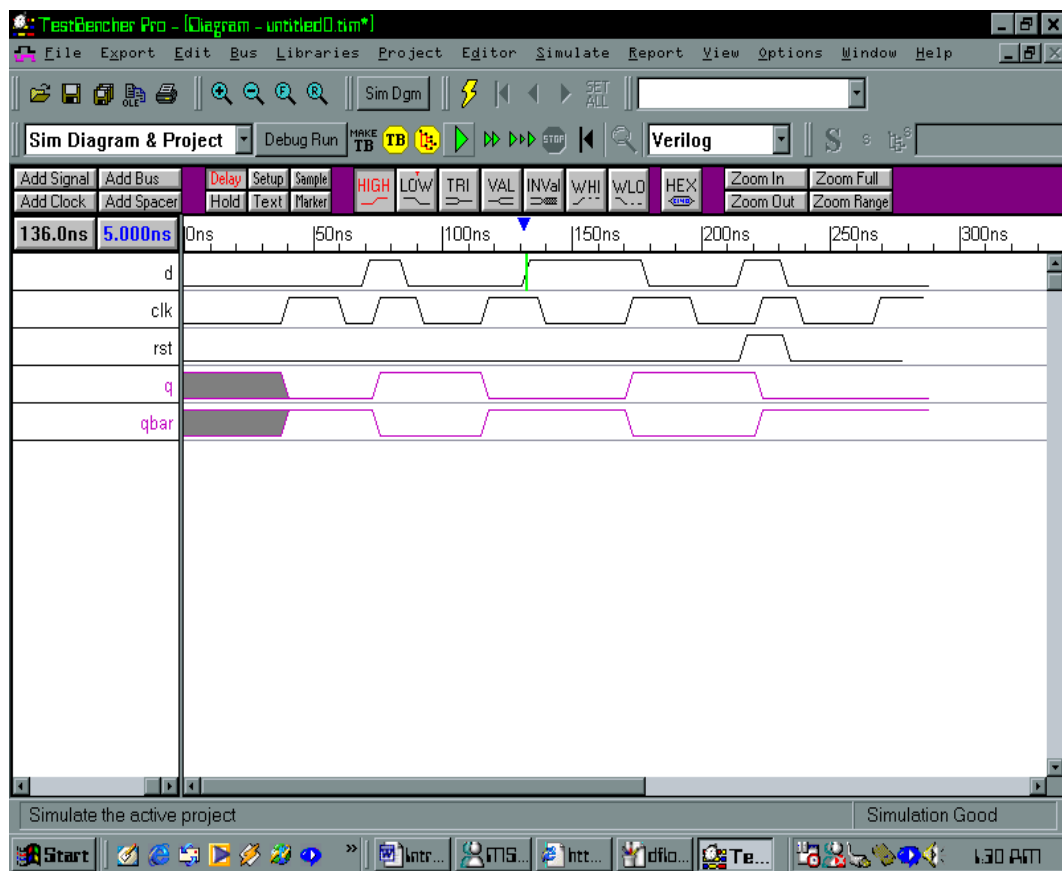


Figure 3.4: Illustration of instantaneous test bench creation

3.5 VERILOG TO VHDL CONVERSION AND VICE VERSA.

Two of the most common HDLs used to describe circuits today are Verilog and VHDL. In a university environment, it is highly likely that in a design team, different members know only one of the HDLs. Rather than waste time and effort to get the entire team to use one single language, we can use syntax mapping software like X-HDL3 to help us toggle between different HDLs. This allows development in both forms of HDLs and the final project can be integrated into either of the languages. Figure 3.5 a show screen capture of the X-hdl3 interface.

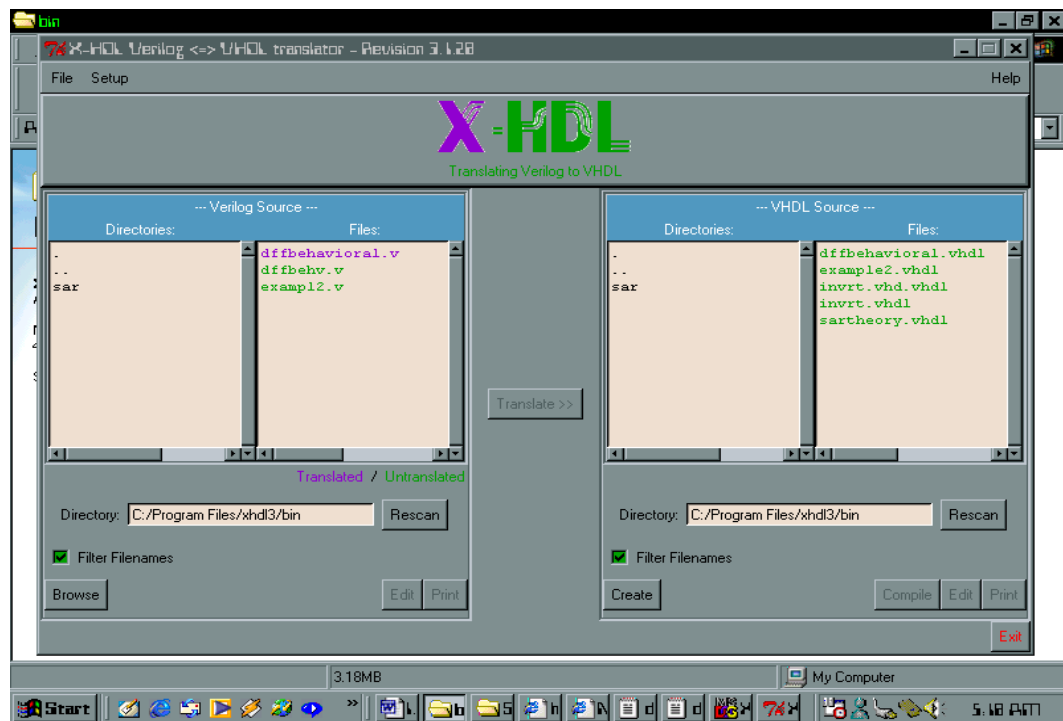


Figure 3.5: X-HDL3 interface for Verilog ↔ VHDL conversion

From a designer's point of view, a behavioral description is a necessary step in the design flow as it forms the basis of the future course of action. While migrating to the next step, RTL code, it is often found that the designer has to start over because some of the behavioral code written is not automatically convertible to RTL (refer section 3.2)

. It would be a good idea to take advantage of software like TestBencher to write behavioral code and use X-HDL3 to make the transition to RTL automated instead of manual. In full custom chips, RTL is followed by a structural description. In the design flow discussed in this document, RTL is avoided, as a working synthesizer to convert RTL to layout is not presently installed at The Ohio State University. The next chapter discusses structural description.

CHAPTER 4

STRUCTURAL DESCRIPTION

A circuit description can be one of three types, a structural description, a functional description (RTL) or a behavioral description. A structural description defines the physical makeup of the circuit, detailing components and the connections between them. It specifies how the gate/transistor level components are interconnected to form a circuit. The design at this stage gets closer to the real world electronic components. The structure of a structural description consists of the I/O port declarations and component instantiations. In most cases, structural descriptions are not made with a particular foundry or process in mind. But for more accurate timing analysis, CAD tools allow structural descriptions to be made with a particular process technology (Sec 5.4), especially if the target chip is an FPGA or an ASIC (section 4.4 and 4.5). In full custom IC design, the design is tied to a process technology at a later stage. Figure 4.1 shows the current status of the design flow up to this chapter.

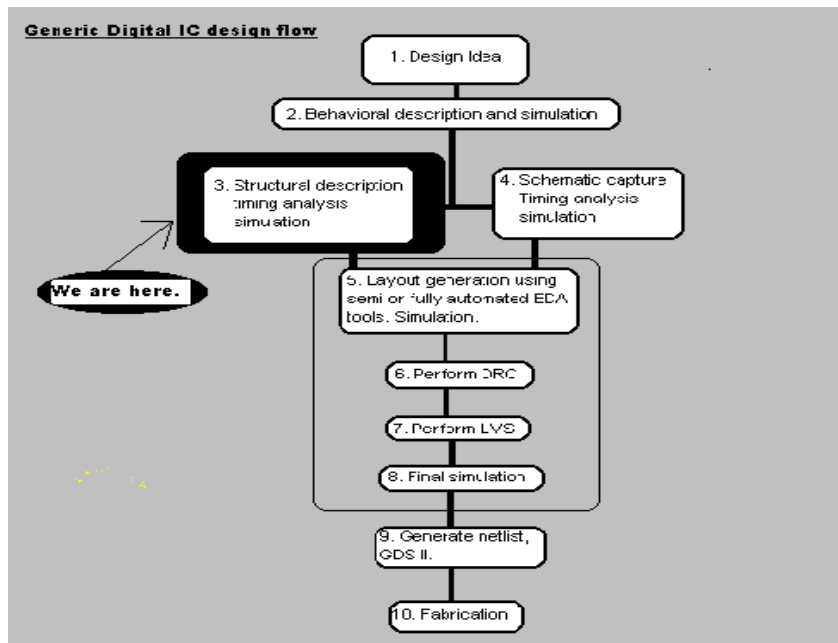


Figure 4.1: current status: structural description

4.1 SIMULATION AND TIMING ANALYSIS

The aim of a structural description is to get a simulation result as close to the behavioral model as possible (Chapter 3). As an example, table 4.1 shows a structural model of a D-flip-flop. Most commercial CAD tools allow users to load delay, timing, and waveform models of HDL primitives that are specific to a process. These libraries are usually not free and must be bought from the vendor.

To avoid extra costs, the modeling is done using a generic default library and the delay models are analyzed for critical paths. Since these default libraries are highly flexible, designers can use user defined delay models, which closely correspond to the real components. The advantage of using this procedure is that the entire circuit can be tested for the worst-case delay without committing to any process technology and is cheaper. Figure 4.2 shows the simulation of a D flip-flop with a generic default library. Figure 4.3 shows the case where setup and hold times are user defined (simulation done using TestBencher).

```
module dff (q,qbar,d,rst,clk);  
input d,clk,rst;  
output q,qbar;  
wire s,r,t,u,rstbar,din;  
not (rstbar,rst);  
and g5 (din,d,rstbar);  
nand g4 (t,r,din);  
nand g3 (r,t,clk,s);  
nand g2 (s,u,clk);  
nand g1 (u,s,t);  
nand g7 (q,qbar,s);  
nand g8 (qbar,q,r);  
endmodule
```

Table 4.1: Structural description of Dff

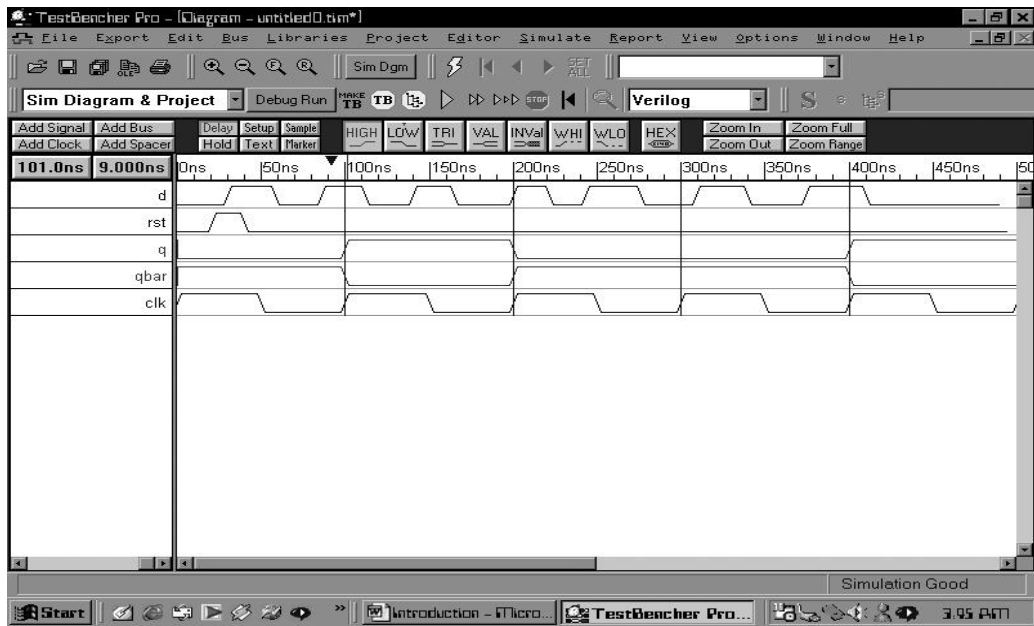


Figure 4.2: simulation of structural dff

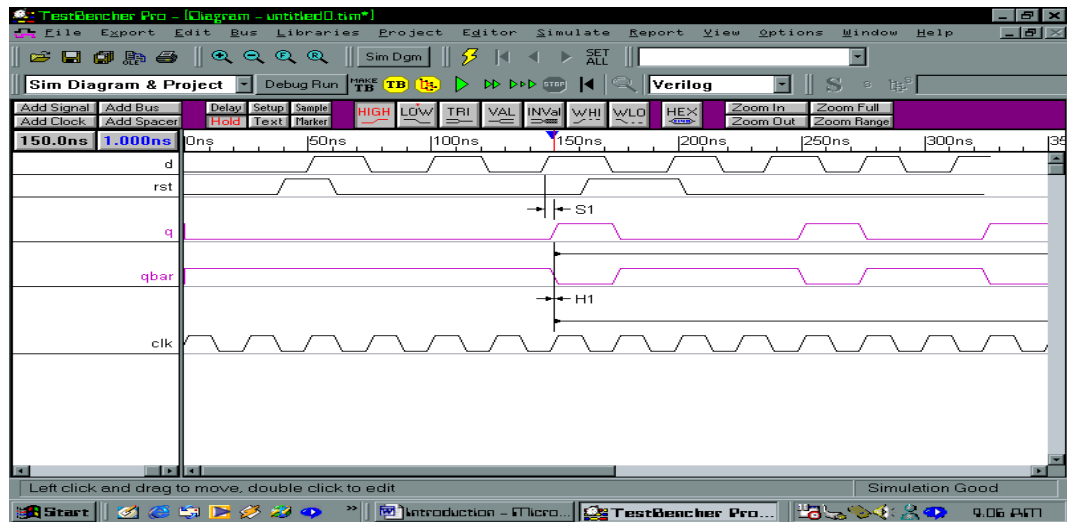


Figure 4.3: User defined setup and hold times in a generic library.

4.2 HIERARCHY

Digital circuitry is inherently hierarchical. Transistors are at the bottom most rung in the hierarchy tree. Logic gates are made up of transistors, flip-flops and latches are made up of logic gates, and more complex circuits are made up of latches and flip-flops and so on. The design in a structural description proceeds in a similar bottom up fashion. The advantage of this approach is the reusability of different logic modules. This is also reflected in the layout phase of the design (Chapter 5) where standard cells, say a D flip-flop are arrayed together to form a register. It is thus very important to maintain this hierarchy with a one to one mapping of standard cells and primitives. Most CAD tools allow easy hierarchical manipulation. If a description at a lower level is changed, the change is automatically reflected at higher levels where that module is used.

Refer to appendix B where a complete structural description of the SAR circuit is described in a hierarchical structural description.

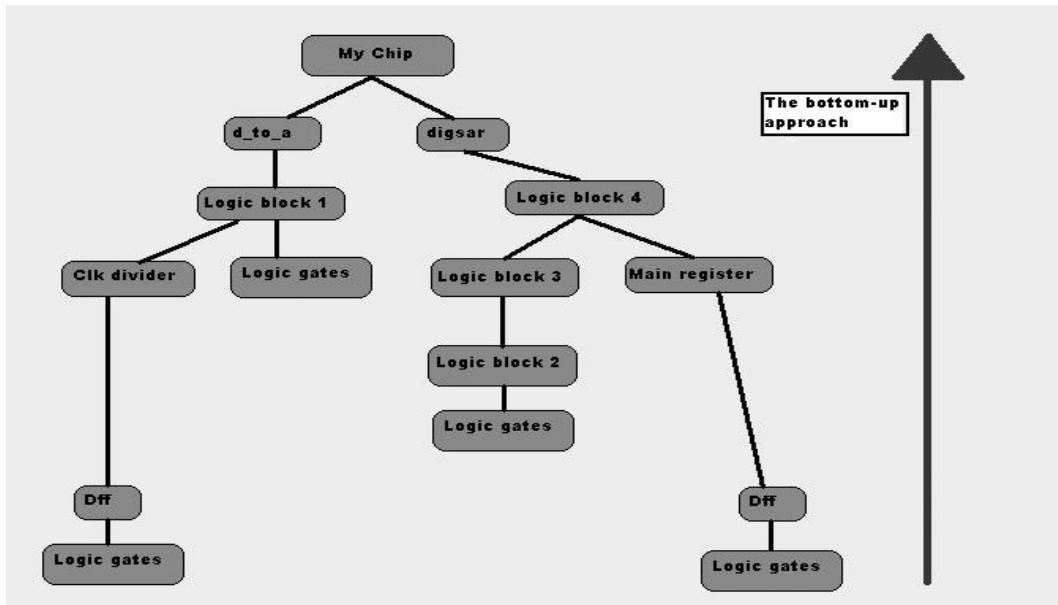


Figure 4.4: hierarchy of the SAR circuit

Given below is a screen capture of TestBencher with 'MyChip' module simulated.

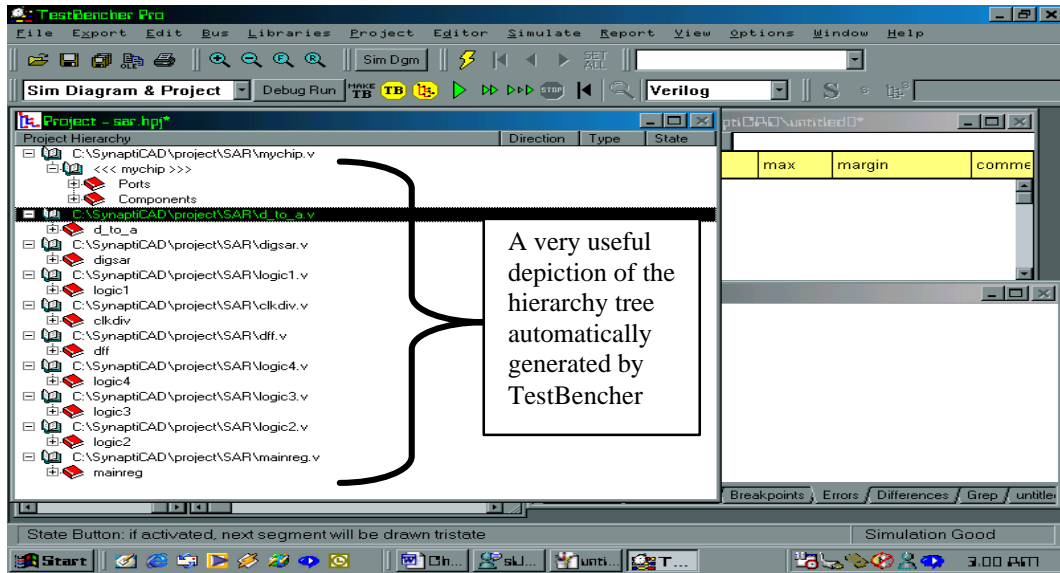


Figure 4.5: Hierarchy in TestBencher

4.3 ANALOG SIMULATION OF STRUCTURAL DESCRIPTION

Most CAD tools allow users to export spice decks and netlists of truly structural descriptions (descriptions that contain only primitives) that can be simulated at an analog level instead of event driven simulations. Analog simulation is needed for more accurate delay and timing analysis. As an example, consider dynamic logic circuits (like zipper and domino logic). Analog simulations are critical for accurate timing analysis of such circuits. The trade off is the simulation time. An analog simulation of the SAR took 3.5 hours to run. Clearly, it is not viable to do this for a large digital system. The usual practice is to simulate small standard cells (like gates, latches and flip-flops) and make sure that they satisfy all the timing requirements. Since the upper hierarchy is made up of these building blocks, it is assumed that the timing requirements will be met.

It is to be noted that in exporting spice decks from HDL descriptions parasitic effects are not extracted. Thus the simulation is only partly accurate. Figures 4.6, 4.7 and 4.8 show the analog simulation of the Verilog description of the D-flip-flop in Microwind. The spice deck was exported to Microwind from TestBencher. Microwind can directly take verilog descriptions and has an inbuilt SPICE simulator.

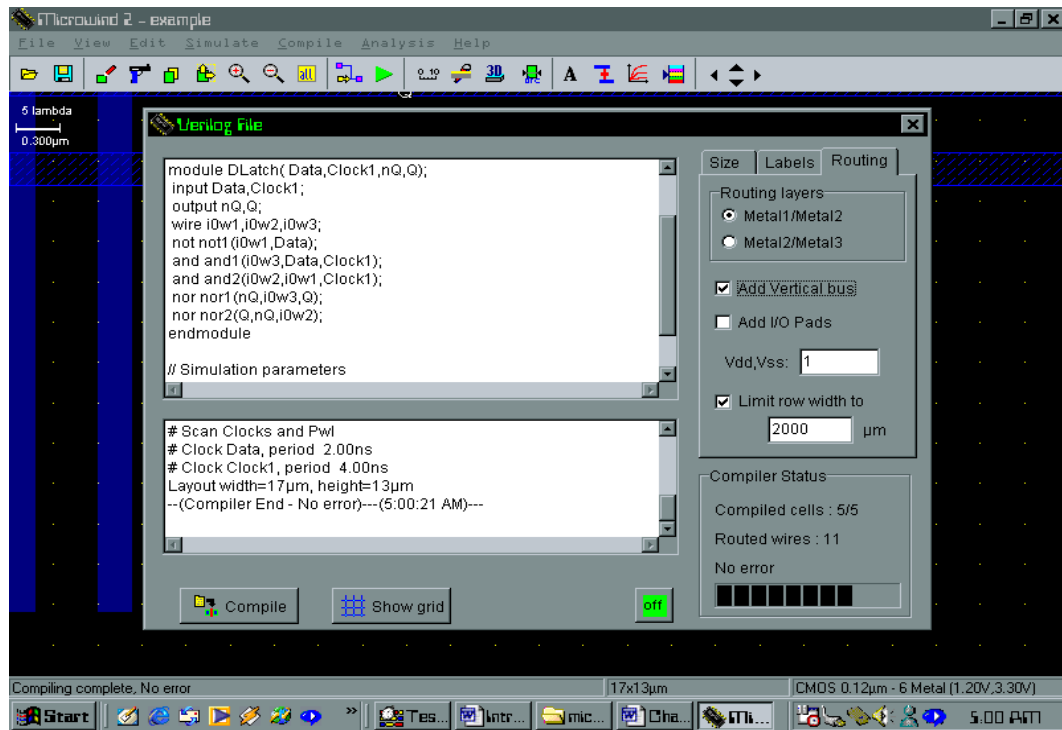


Figure 4.6: Screen capture of Microwind showing direct Verilog file input

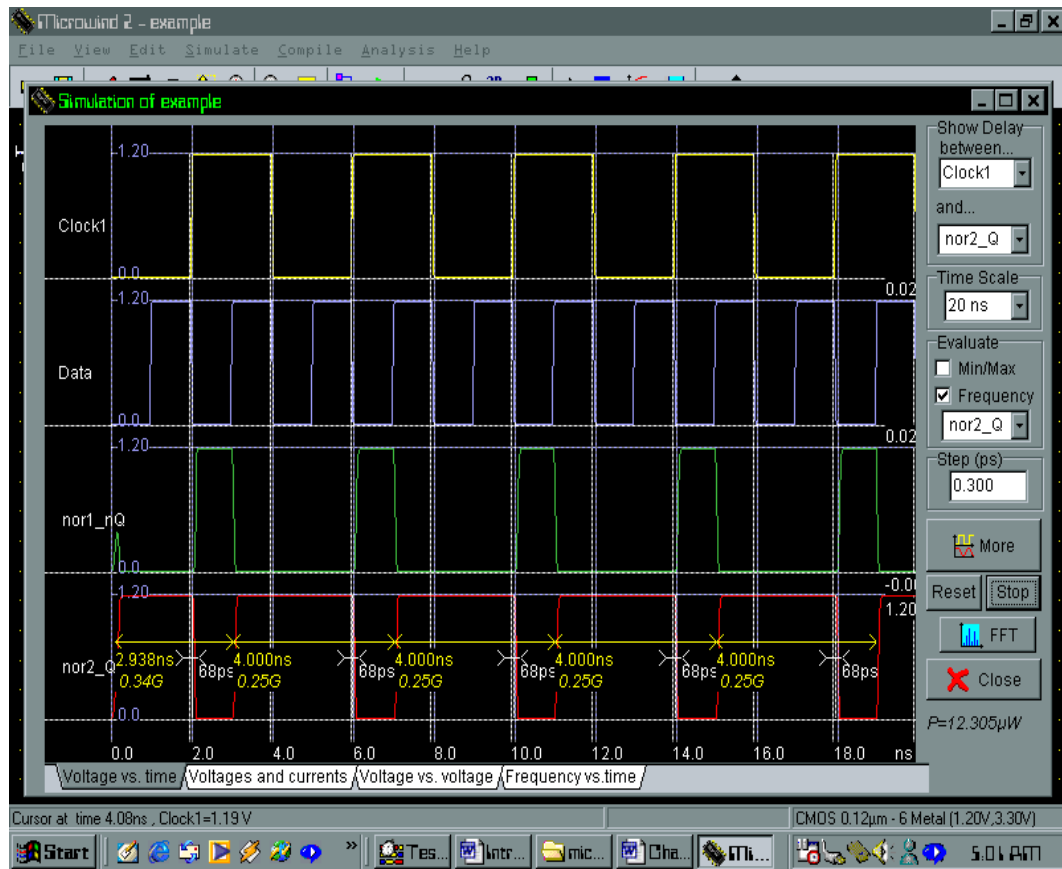


Figure 4.7: Microwind SPICE simulation 1

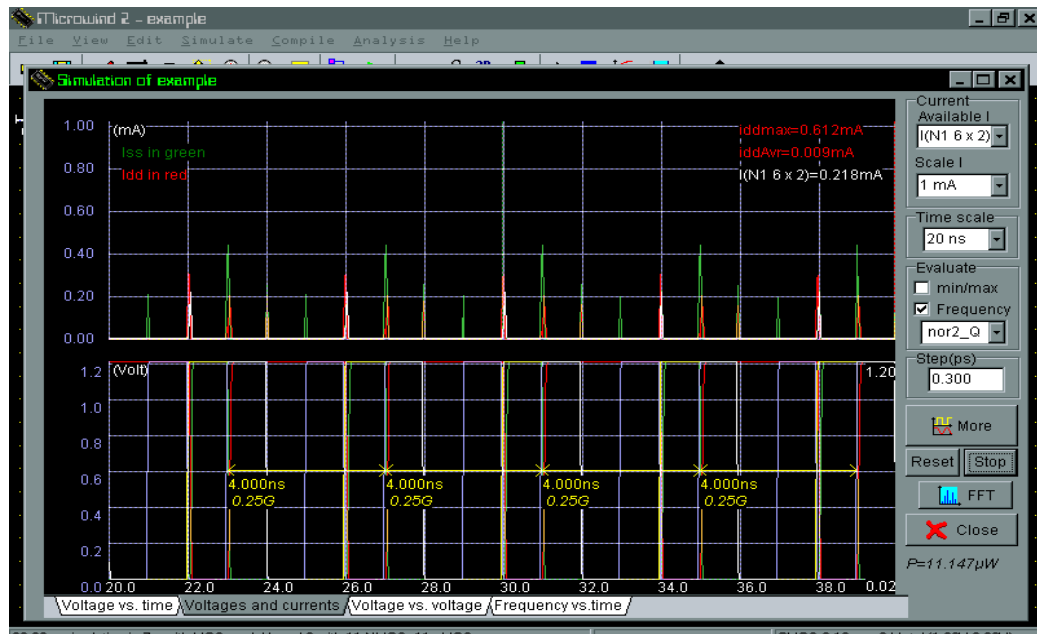


Figure 4.8: Microwind SPICE simulation 2

4.4 STRUCTURAL DESCRIPTION TO FPGA

If the target chip happens to be digital, it often synthesized into an FPGA. An FPGA is a programmable chip that can be treated as a prototype of the full custom IC that is to be built. The advantage of getting an FPGA synthesized before proceeding towards a full custom chip is that it is possible to perform an early functional debug. Though FPGA synthesis involves money, it would payoff at a later stage as valuable information can be gathered before investing time and money in a custom IC process.

There are many vendors in the market that cater to FPGAs. Two of the most notable ones are Xilinx and Altera. Figure 4.9 shows the design flow adopted for FPGA synthesis using Mycad.

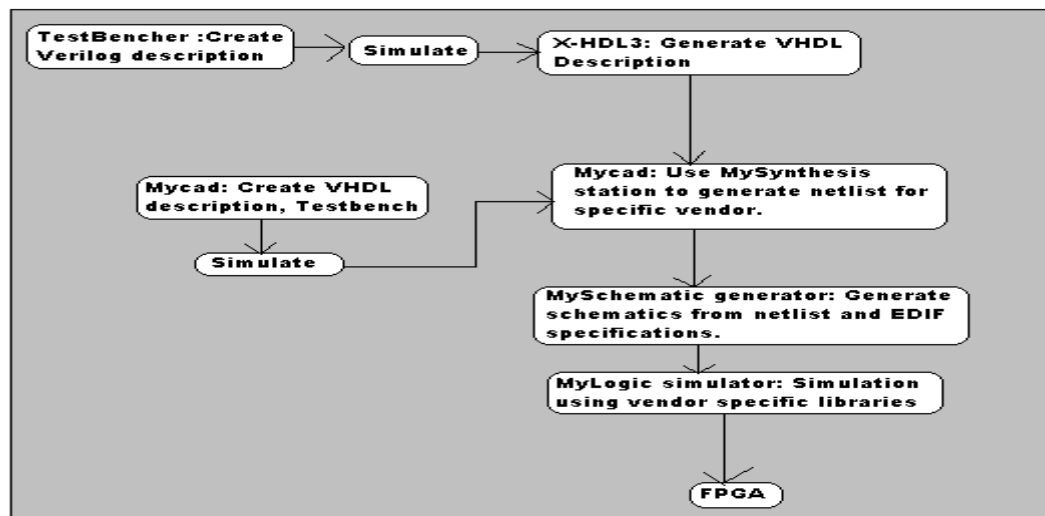


Figure 4.9: FPGA Synthesis Design flow Using MyCad

Table 4.2 illustrates the implementation of above design flow for a simple 8 bit adder.

```

module onebadd (sum,cout,a,b,cin);
input a, b,cin;
output sum,cout;
assign sum=a^b^cin;
assign cout=cin & (a/b) | a&b;
endmodule
module eightbadd (sum,cout,a,b,cin);
input [7:0]a;
input [7:0]b;
input cin;
wire [6:0]co;
output [7:0]sum;
output cout;
onebadd (sum[0],co[0],a[0],b[0],cin);
onebadd (sum[1],co[1],a[1],b[1],co[0]);
onebadd (sum[2],co[2],a[2],b[2],co[1]);
onebadd (sum[3],co[3],a[3],b[3],co[2]);
onebadd (sum[4],co[4],a[4],b[4],co[3]);
onebadd (sum[5],co[5],a[5],b[5],co[4]);
onebadd (sum[6],co[6],a[6],b[6],co[5]);
onebadd (sum[7],cout,a[7],b[7],co[6]);
endmodule

```

--VHDL Files generated through X-HDL3

```

-----
--
-- VHDL file generated by X-HDL - Revision 3.1.28
-- Thu May 17 17:04:47 2001
--
--   Input file      : C:/Program Files/xhdl3/bin/onebadd.v
--   Design name     : onebadd

```

```

ENTITY onebadd IS
PORT (

```

Table 4.2: 8 bit adder structural descriptions in VHDL and Verilog (continued)

Table 4.2: (continued)

```

sum          : OUT bit;
cout         : OUT bit;
a            : IN bit;
b            : IN bit;
cin          : IN bit;
clk          : IN bit);
END onebadd;

```

ARCHITECTURE translated OF onebadd IS

```

SIGNAL sum_xhdl1      : bit;
SIGNAL cout_xhdl2    : bit;

BEGIN
  sum <= sum_xhdl1;
  cout <= cout_xhdl2;
  sum_xhdl1 <= (a XOR b) XOR cin ;
  cout_xhdl2 <= (cin AND (a OR b)) OR (a AND b) ;

END translated;

```

8 bit adder file generated by X-HDL3

```

-----
---
--
-- VHDL file generated by X-HDL - Revision 3.1.28
-- Thu May 17 18:51:03 2001
--
-- Input file      : C:/Program Files/xhdl3/bin/eightbadd.v
-- Design name     : eightbadd
ENTITY eightbadd IS
  PORT (
    sum          : OUT bit_vector(7 DOWNTO 0);
    cout         : OUT bit;

```

Table 4.2: (continued)

Table 4.2: (continued)

```

a          : IN bit_vector(7 DOWNTO 0);
b          : IN bit_vector(7 DOWNTO 0);
cin        : IN bit);

```

END eightbadd;

ARCHITECTURE translated OF eightbadd IS

```

COMPONENT onebadd
  PORT (
    sum          : OUT bit;
    cout         : OUT bit;
    a            : IN bit;
    b            : IN bit;
    cin          : IN bit;
    clk          : IN bit);
END COMPONENT;

```

```

SIGNAL co          : bit_vector(6 DOWNTO 0);
SIGNAL port_xhdl4  : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl5  : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl6  : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl7  : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl9  : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl10 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl11 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl12 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl13 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl15 : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl16 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl17 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl18 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl19 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl21 : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl22 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl23 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl24 : bit_vector(1 DOWNTO 0);

```

Table 4.2: (continued)

Table 4.2: (continued)

```

SIGNAL port_xhdl25 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl27 : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl28 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl29 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl30 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl31 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl33 : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl34 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl35 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl36 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl37 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl39 : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl40 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl41 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl42 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl43 : bit_vector(2 DOWNTO 0);
SIGNAL port_xhdl45 : bit_vector(9 DOWNTO 0);
SIGNAL port_xhdl46 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl47 : bit_vector(1 DOWNTO 0);
SIGNAL port_xhdl48 : bit_vector(2 DOWNTO 0);
SIGNAL sum_xhdl1 : bit_vector(7 DOWNTO 0);
SIGNAL cout_xhdl2 : bit;

```

BEGIN

```

sum <= sum_xhdl1;
cout <= cout_xhdl2;
port_xhdl4 <= sum_xhdl1(0);
port_xhdl5 <= co(0);
port_xhdl6 <= a(0);
port_xhdl7 <= b(0);
xhdl3 : onebadd
  PORT MAP (
    sum => port_xhdl4,
    cout => port_xhdl5,
    a => port_xhdl6,
    b => port_xhdl7,
    cin => cin);

```

Table 4.2: (continued)

Table 4.2: (continued)

```

port_xhdl9 <= sum_xhdl1(1);
port_xhdl10 <= co(1);
port_xhdl11 <= a(1);
port_xhdl12 <= b(1);
port_xhdl13 <= co(0);
xhdl8 : onebadd
  PORT MAP (
    sum => port_xhdl9,
    cout => port_xhdl10,
    a => port_xhdl11,
    b => port_xhdl12,
    cin => port_xhdl13);

```

```

port_xhdl15 <= sum_xhdl1(2);
port_xhdl16 <= co(2);
port_xhdl17 <= a(2);
port_xhdl18 <= b(2);
port_xhdl19 <= co(1);
xhdl14 : onebadd
  PORT MAP (
    sum => port_xhdl15,
    cout => port_xhdl16,
    a => port_xhdl17,
    b => port_xhdl18,
    cin => port_xhdl19);

```

```

port_xhdl21 <= sum_xhdl1(3);
port_xhdl22 <= co(3);
port_xhdl23 <= a(3);
port_xhdl24 <= b(3);
port_xhdl25 <= co(2);
xhdl20 : onebadd
  PORT MAP (
    sum => port_xhdl21,
    cout => port_xhdl22,
    a => port_xhdl23,
    b => port_xhdl24,
    cin => port_xhdl25);

```

Table 4.2: (continued)

Table 4.2: (continued)

```

port_xhdl27 <= sum_xhdl1(4);
port_xhdl28 <= co(4);
port_xhdl29 <= a(4);
port_xhdl30 <= b(4);
port_xhdl31 <= co(3);
xhdl26 : onebadd
  PORT MAP (
    sum => port_xhdl27,
    cout => port_xhdl28,
    a => port_xhdl29,
    b => port_xhdl30,
    cin => port_xhdl31);

```

```

port_xhdl33 <= sum_xhdl1(5);
port_xhdl34 <= co(5);
port_xhdl35 <= a(5);
port_xhdl36 <= b(5);
port_xhdl37 <= co(4);
xhdl32 : onebadd
  PORT MAP (
    sum => port_xhdl33,
    cout => port_xhdl34,
    a => port_xhdl35,
    b => port_xhdl36,
    cin => port_xhdl37);

```

```

port_xhdl39 <= sum_xhdl1(6);
port_xhdl40 <= co(6);
port_xhdl41 <= a(6);
port_xhdl42 <= b(6);
port_xhdl43 <= co(5);
xhdl38 : onebadd
  PORT MAP (
    sum => port_xhdl39,
    cout => port_xhdl40,
    a => port_xhdl41,
    b => port_xhdl42,
    cin => port_xhdl43);

```

Table 4.2: (continued)

Table 4.2: (continued)

```
port_xhdl45 <= sum_xhdl1(7);  
port_xhdl46 <= a(7);  
port_xhdl47 <= b(7);  
port_xhdl48 <= co(6);  
xhdl44 : onebadd
```

```
  PORT MAP (  
    sum => port_xhdl45,  
    cout => cout_xhdl2,  
    a => port_xhdl46,  
    b => port_xhdl47,  
    cin => port_xhdl48);
```

```
END translated;
```

Figure 4.10 shows MyCad's MySynthesizer synthesizing the VHDL code (refer table 4.2) for the Xilinx Spartan FPGA library.

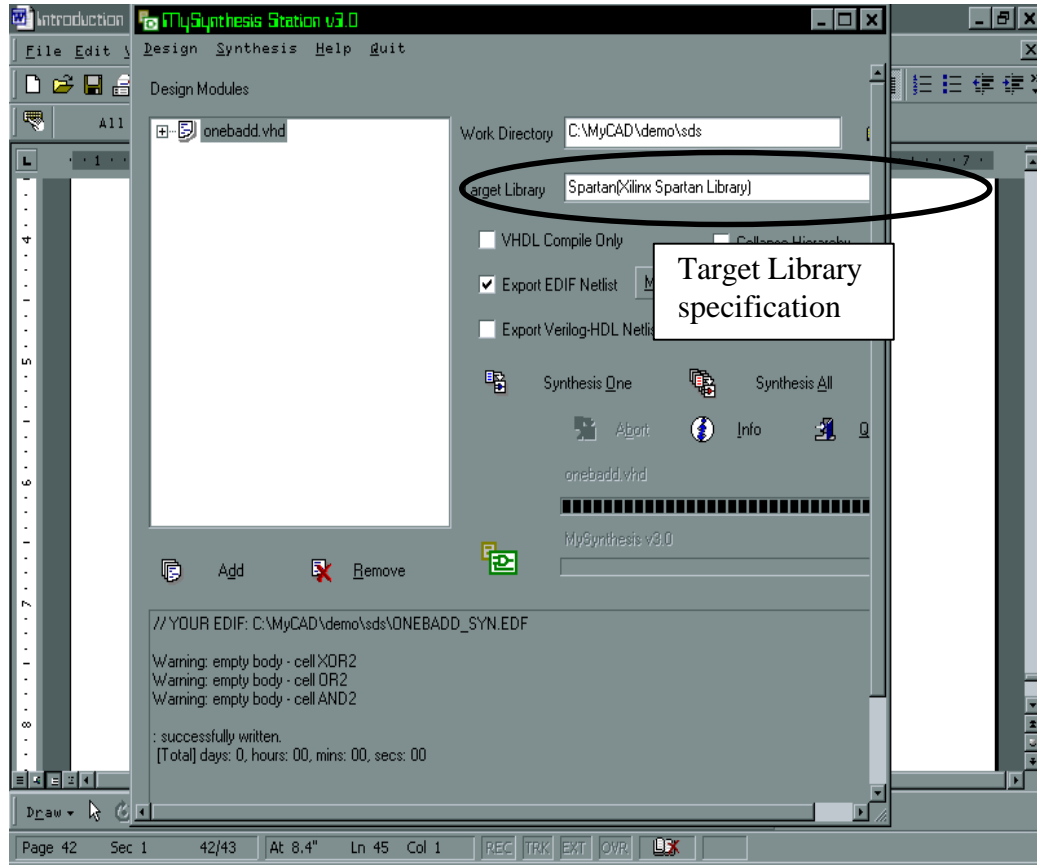


Figure 4.10 : MyCad's FPGA synthesizer

Figure 4.11 shows the schematic being generated for one bit adder. Figure 4.12 shows the schematic generated for the 8 bit adder.

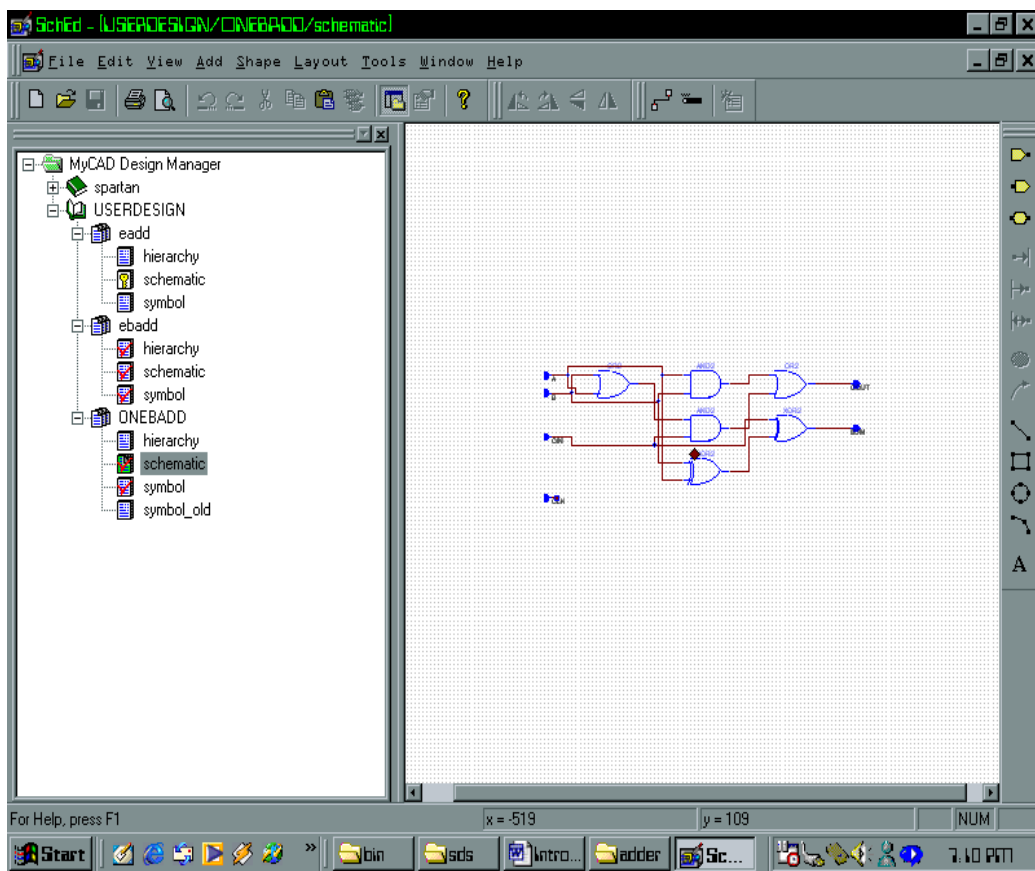


Figure 4.11: One bit adder FPGA

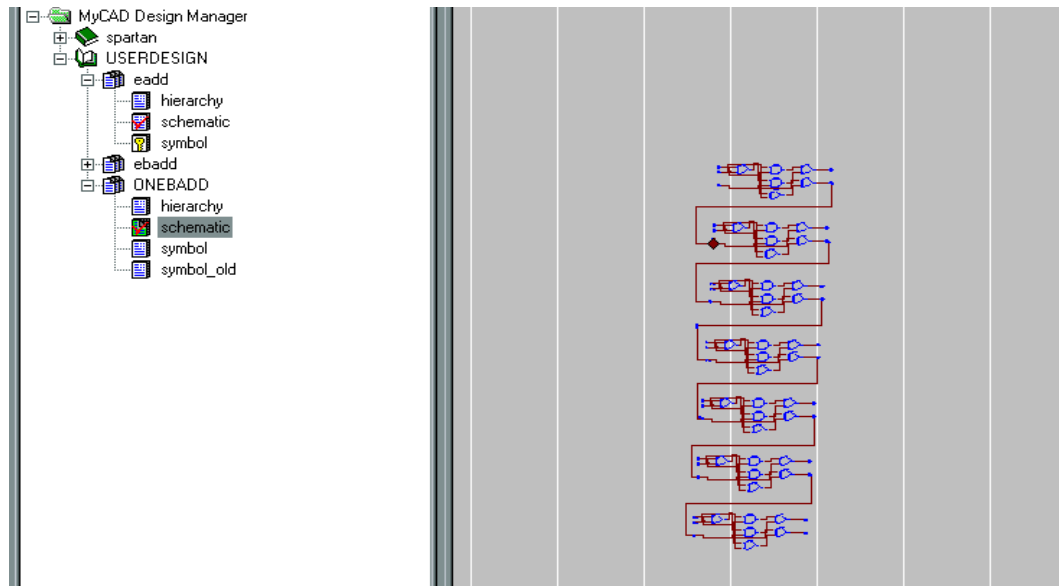


Figure 4.12: 8 Bit adder FPGA

4.5 STRUCTURAL DESCRIPTION TO ASIC USING RENOIR (MENTOR GRAPHICS)

The difference between an ASIC and an FPGA is that FPGA solutions are programmable and cheaper in small quantities (roughly in tens of numbers). An ASIC solution is a more expensive investment but would prove cheaper if the chips are made in large quantities. Since most chips are designed to be mass-produced, the general practice is to obtain an FPGA solution for preliminary testing and then

synthesize the design for an ASIC. In a computer motherboard, most of the chips that we see are ASICs.

ASIC and FPGA design flows are usually the same. In Renoir, the Design Flow is exactly the same for ASIC and FPGA. Figure 4.13 shows the ASIC design flow in Renoir.

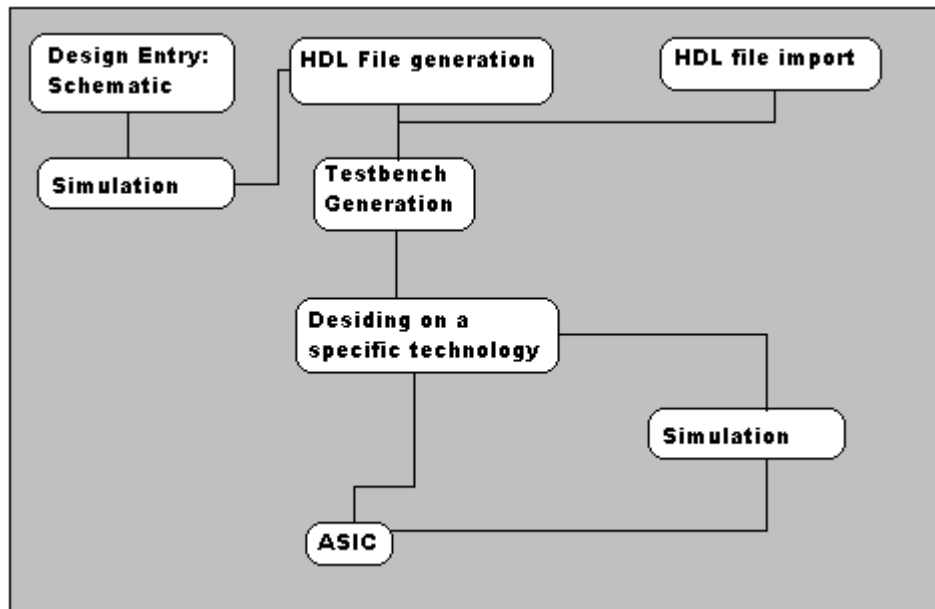


Figure 4.13: Renoir ASIC design Flow.

The most critical stage in chip design is the transition from a structural description a full custom chip layout. The next chapter discusses the final step in the design flow, IC layout.

CHAPTER 5

IC LAYOUT, DRC AND LVS

IC Layout is the term used to describe the geometry of the various layers of materials that go into the making of a chip. This is a description of how the chip would look if viewed under a microscope. Prior to the layout phase, the design is usually not committed to a process. But before starting the layout phase, we need to specify what fabrication plant the design is aimed at. The reason being, each fabrication process has its own set of design rules and layout is something that cannot be changed very easily. Due rigor and rigidity of the layout process, we need an industrially accepted tool to work with. There are many inexpensive CAD tools that are capable of editing layouts. But the problem with such tools arises when we need to tie the chip to a particular fabrication technology. Since most vendors offer design kits that are customized for a few well-known tools, other CAD tools cannot be effectively used to generate chips for that foundry.

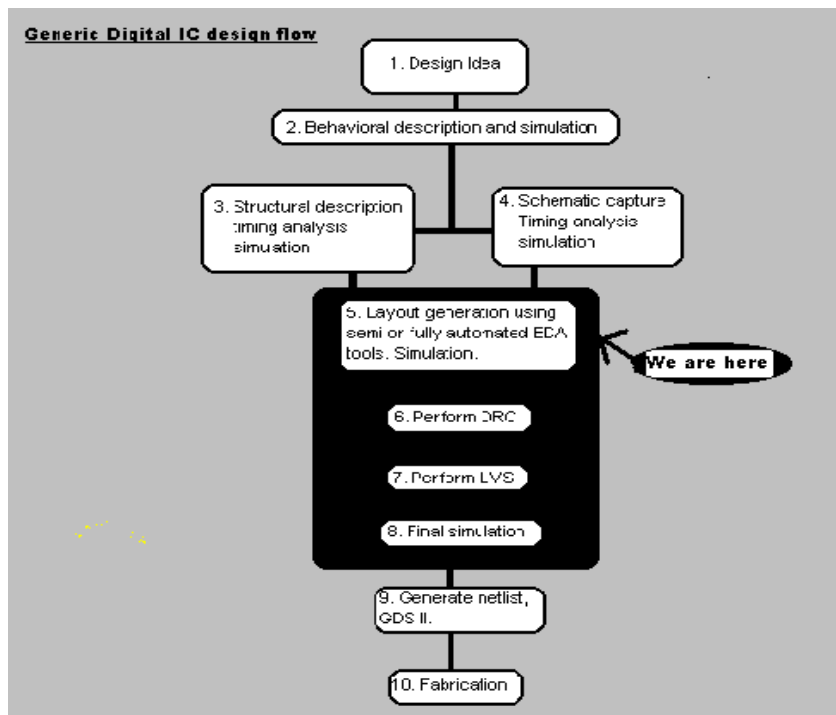


Figure 5.1: Current status: Layout, DRC, LVS

5.1 CADENCE AND MENTOR GRAPHICS VERSUS OTHER TOOLS

Cadence and Mentor Graphics design suites are usually the default tools used by universities to tape-out chips. Cadence and MentorGraphics are also the most widely accepted CAD tools in the industry. They are capable of generating complete design flows within themselves (design entry to CIF/GDSII). The

disadvantage of using such tools is their cost and inflexibility. As an example, they do not have an automatic test bench generation capabilities, Verilog to VHDL conversion and vice versa, operating system independence etc. Sometimes, private design groups cannot afford to purchase entire design suites of such tools.

In places where a synthesizer is absent, a lot of time is wasted in manually placing and routing standard cells. The question now is, how to make this design flow faster and cheaper? One alternative is to buy a synthesizer. But this would mean a lot of money and the design flow would no longer be cost effective. The other alternative is to use free software like Electric to produce a design.

The main problem with lesser-known tools is their reliability. As an example, software like Microwind can read and write CIF files but the DRC is never satisfied. Figure 5.2 shows a Dflipflop designed in Microwind but when imported into cadence it never satisfies the Design Rule Checks (Figure 5.3)

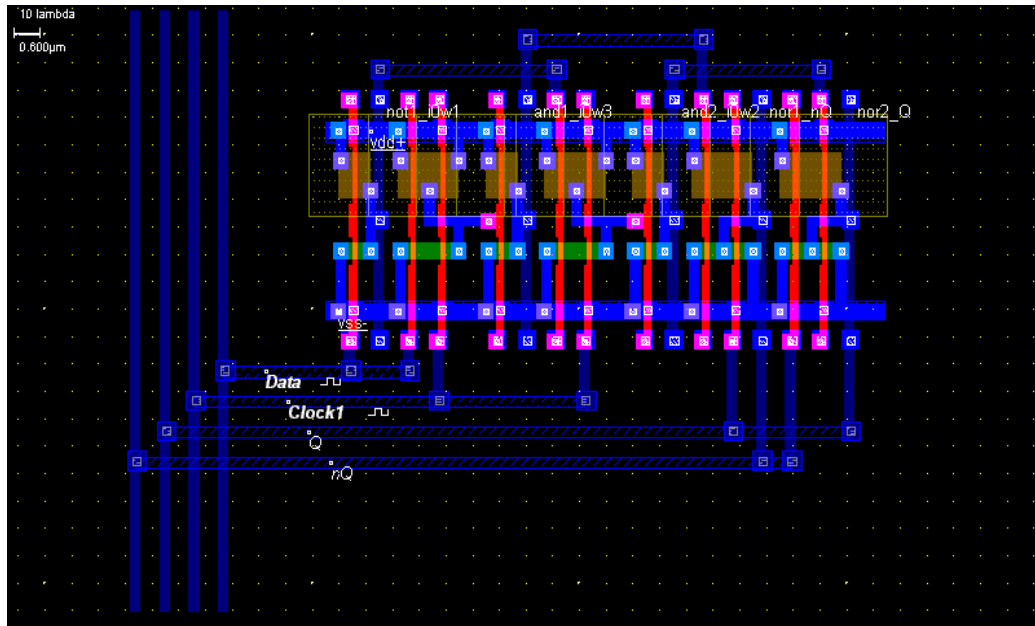


Figure 5.2: Dflipflop synthesized by Microwind

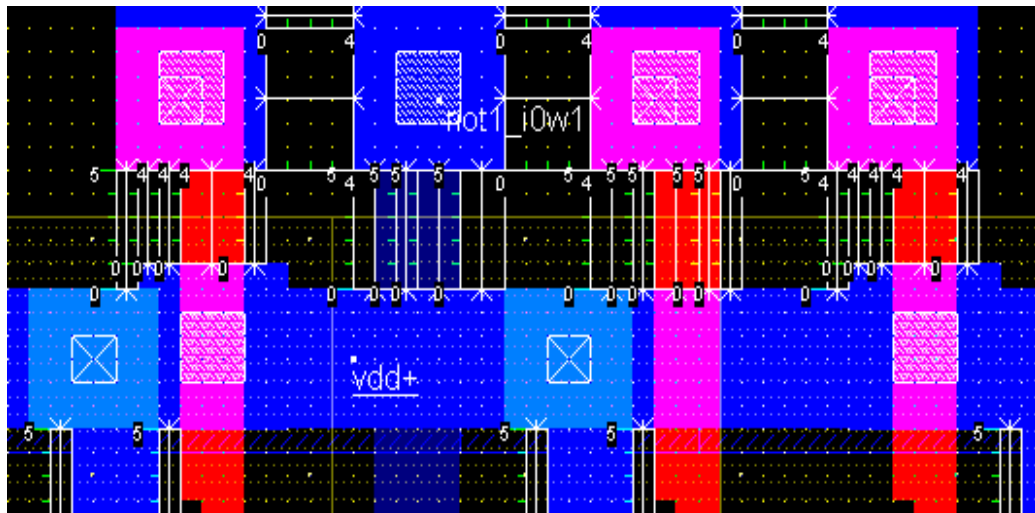


Figure 5.3: DRC failure of Microwind's Dff in Cadence

However, not all free CAD tools are unreliable. The rest of this chapter deals with Electric, which is a free CAD tool suite, and attempts to evolve a reliable design flow using the same.

5.2 WHY CADENCE AT THE LAST STEP?

As mentioned before, Cadence is one of the most well accepted tools by the industry. Fabrication plants supply their standard cell libraries (for a price) customized for such tools. Thus, it becomes a necessity to use Cadence like tools for verifying the integrity of lesser-known tools. Fortunately, there are some standards, which are followed in the industry (like CIF and GDSII) that can be read in by other cheaper software and processed. The acid test would be to import the design back into Cadence and ensure that all of the design passes the DRC test. The design flow attempted in this chapter for generating fast reliable layouts is shown in figure 5.4.

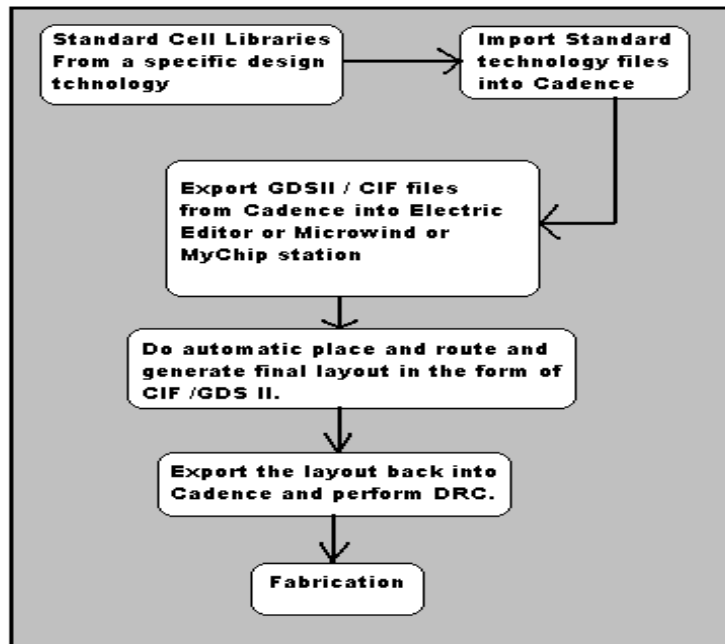


Figure 5.4: Design flow for layout generation

5.3 STARTING A LAYOUT FOR MOSIS

Most of the chips designed in universities are targeted at MOSIS. MOSIS is a low-cost prototyping and small-volume production service for custom and semi-custom VLSI integrated circuits. MOSIS has an excellent educational institution program that makes it ideal for universities. Refer to www.mosis.org for more information. MOSIS places transistor simulation models and the electrical

properties of the other integrated circuit structures online. These can be used in circuit simulators to model the performance of designs before their fabrication.

MOSIS provides the information given in table 5.1 to ensure that the simulations and the design done by the design group conform to the real parameters for that technology. Table 5.1 shows the steps needed create a design for MOSIS. Figure 5.5 shows the design flow recommended by MOSIS.

Resources provided by MOSIS.

- 1. Transistor SPICE models and electrical parameters.*
- 2. Libraries as available, with cell timing and functional models*
- 3. Deliver bonded, packaged, fabricated parts to customers*

Table 5.1: Resources provided by MOSIS for designers

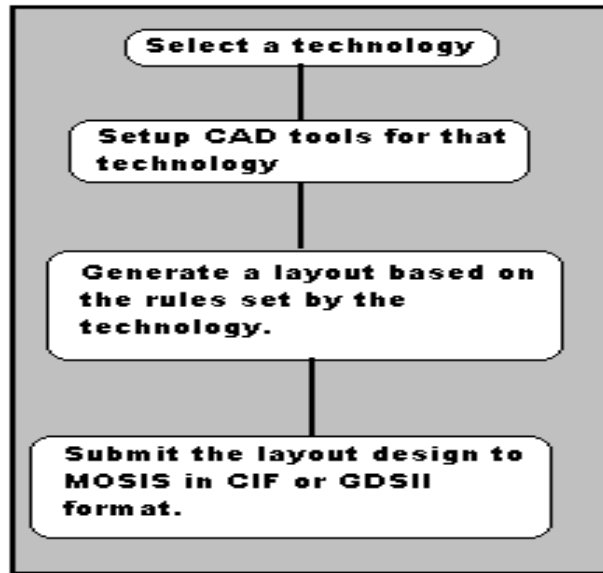


Figure 5.5: MOSIS recommended design flow.

The rest of this chapter deals with achieving the design flow in figure 5.5 using Electric. Once this design flow is achieved, an IC can be fabricated through MOSIS.

5.4 SELECTING A PROCESS TECHNOLOGY

Different vendors offer different IC technologies with different characteristics (as, type or feature size) and at different costs. In general, the smaller the feature size of the technology, the faster its transistors are and more

circuitry can fit in the same area, but the cost of fabrication increases. The various vendors that are available are listed online in MOSIS's website.

5.5 SCMOS VERSUS SPECIFIC VENDOR TECHNOLOGY

SCMOS stands for Scalable CMOS. The advantage of using this technology for layout generation is that the design can be done using a general feature size λ . At the end of the layout phase, the design can be 'blown up' or 'shrunk' by substituting different values of λ to fit a particular process technology. But things are not as transparent as they look. Most of the times the layout fails the DRC as design rules are not always linear. The problem is partially solved when the design is 'blown up', but this means that the transistor size has increased and the chip might be slower than expected and maybe too large for the pad frame. In this document, SCMOS_SUBM at $\lambda=0.30$ ([AMI 0.80](#)) library will be used (because it is free). It should be noted that the same methodology followed here could be applied to any vendor specific technology. Table 5.2 enumerates the necessary features that must be provided by a vendor. Libraries are usually sold in the LEF format. Figures 5.6 and 5.7 show a sample of a cell in the AMI 0.8 technology. The cell shown is a 2 x 2 AND OR Gate.

Vendor specific technology Libraries must include

1. *A list of standard cells (Ex. Primitive gates, multiplexors, buffers etc)*
2. *Schematic description with a spice deck file.*
3. *Verilog/VHDL description.*
4. *Truth table or state diagram.*
5. *Node Capacitances.*
6. *Delay Characteristics*
7. *GDS II / CIF plot*

Table 5.2: Features to be supplied by vendors for their libraries

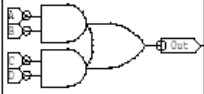

| 2X2 Input AND-OR | | | | AO22 | | | | | | | | | | | | | | | | | | | | | | |
|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------|------------------------|--------------------|-----|-----|---|---|---|---|---|---|---|---|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--------|---|-------|---|-------|---|-------|---|-------|
| Description: 2 X 2 Input AND-OR Gate | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Library: Tanner mAMIs05DL | Primitive Set: | Tanner.SCMOS.Cells Tanner.TIB.Samples | | | | | | | | | | | | | | | | | | | | | | | | |
| Schematic: S-Edit | File: | TannerLb\scmos\scmos.sdb | | | | | | | | | | | | | | | | | | | | | | | | |
| Mask layout: L-Edit | Module: | AO22 | | | | | | | | | | | | | | | | | | | | | | | | |
| | File: | TannerLb\scmos\scmos.tdb | | | | | | | | | | | | | | | | | | | | | | | | |
| | Cell: | AO22 | | | | | | | | | | | | | | | | | | | | | | | | |
| Mapping Macros: GateSim: | TannerLb\nettran\scmos\scms2sim.mac | | | | | | | | | | | | | | | | | | | | | | | | | |
| L-Edit/SPR: | TannerLb\nettran\scmos\scms2tpr.mac | | | | | | | | | | | | | | | | | | | | | | | | | |
| Logic Symbol | Truth Table | | | Capacitance | | | | | | | | | | | | | | | | | | | | | | |
|  | <table border="1" style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th>A*B</th> <th>C*D</th> <th>Out</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>X</td> <td>1</td> </tr> <tr> <td>X</td> <td>1</td> <td>1</td> </tr> </tbody> </table> | | | A*B | C*D | Out | 0 | 0 | 0 | 1 | X | 1 | X | 1 | 1 | <table border="1" style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th></th> <th>Cl(fF)</th> </tr> </thead> <tbody> <tr> <td>A</td> <td>6.953</td> </tr> <tr> <td>B</td> <td>6.953</td> </tr> <tr> <td>C</td> <td>6.953</td> </tr> <tr> <td>D</td> <td>6.953</td> </tr> </tbody> </table> | | Cl(fF) | A | 6.953 | B | 6.953 | C | 6.953 | D | 6.953 |
| A*B | C*D | Out | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | X | 1 | | | | | | | | | | | | | | | | | | | | | | | | |
| X | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | |
| | Cl(fF) | | | | | | | | | | | | | | | | | | | | | | | | | |
| A | 6.953 | | | | | | | | | | | | | | | | | | | | | | | | | |
| B | 6.953 | | | | | | | | | | | | | | | | | | | | | | | | | |
| C | 6.953 | | | | | | | | | | | | | | | | | | | | | | | | | |
| D | 6.953 | | | | | | | | | | | | | | | | | | | | | | | | | |
| Height | Width | Area | Equivalent Gate | Drive | | | | | | | | | | | | | | | | | | | | | | |
| 53.3 | 57.3 | 3071.3 ² | 2.6 | 1X | | | | | | | | | | | | | | | | | | | | | | |
| Logic Equation | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Out = (A × B) + (C × D) | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Delay Characteristics: | | | | | | | | | | | | | | | | | | | | | | | | | | |
| $T_{pd} = t_0 + \frac{dt}{dc} \times C_L$ | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Tpd0 → 1.....135 + 780 × C[OUT] | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Tpd1 → 0.....92 + 790 × C[OUT] | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | MOSIS AMI 0.5μ – mAMIs05DL Scalable Digital Standard Cell Library | | Rev. A AO22 | Page 1 of 4 | | | | | | | | | | | | | | | | | | | | | | |

Figure 5.6: AMI 0.8 technology 2 x 2 AND OR Gate

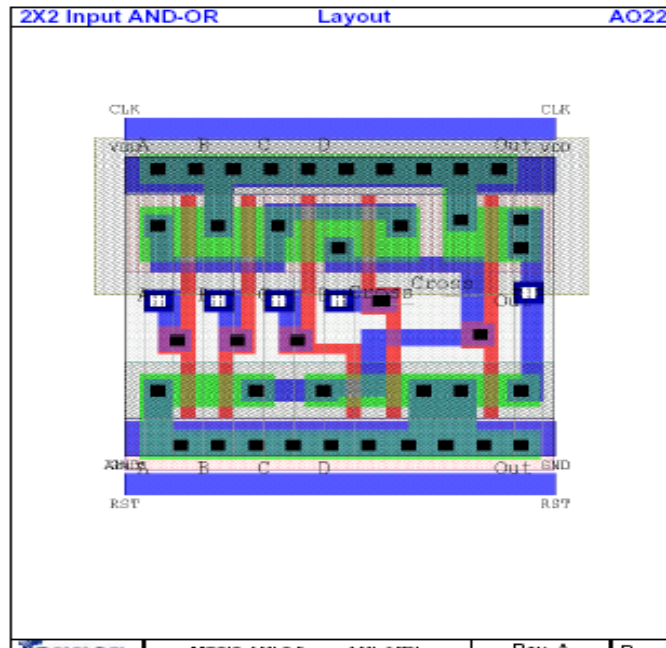


Figure 5.7: AMI 0.8 technology 2 x 2 AND OR Gate GDS II plot.

5.6 LAYOUT FORMATS

MOSIS accepts submissions in CIF and GDS II formats. CIF is an older format for describing layer geometry in ASCII. CIF files are stored in ASCII format and are readable by humans. The IC community, because of accuracy issues is slowly phasing out CIF (for more information refer to www.rulabinsky.com/cavd/text/chap).

The most popular format for interchange at present is the Calma **GDS II** stream format (GDS II is a trademark of Calma Company, a wholly owned subsidiary of General Electric Company, U.S.A. and Cadence). Although Calma has updated the format with the development of their CAD systems, they have maintained backward compatibility so that no GDS II files become obsolete. This is important because GDS II is a binary format that makes assumptions about integer and floating-point representations.

This document will deal with both the GDS II and CIF formats. The conversion from one format to another is not an issue since most CAD tools are capable of converting a layout from GDS II to CIF and vice versa.

5.7 SETTING UP CAD TOOLS

The scope of this document is to find an alternative to the already existing design flow. The CAD tool that is going to be used for layout processing is Electric (www.staticfreesoft.com). The most compelling fact about this software is that it is open sourced, capable of place and route, analog simulation and event based digital simulation. It has also been used for many successful IC tape outs from design groups in Sun Microsystems. Figure 5.8 shows a screen capture of Electric. Table 5.3 shows the verification steps that need to be followed before starting a layout.

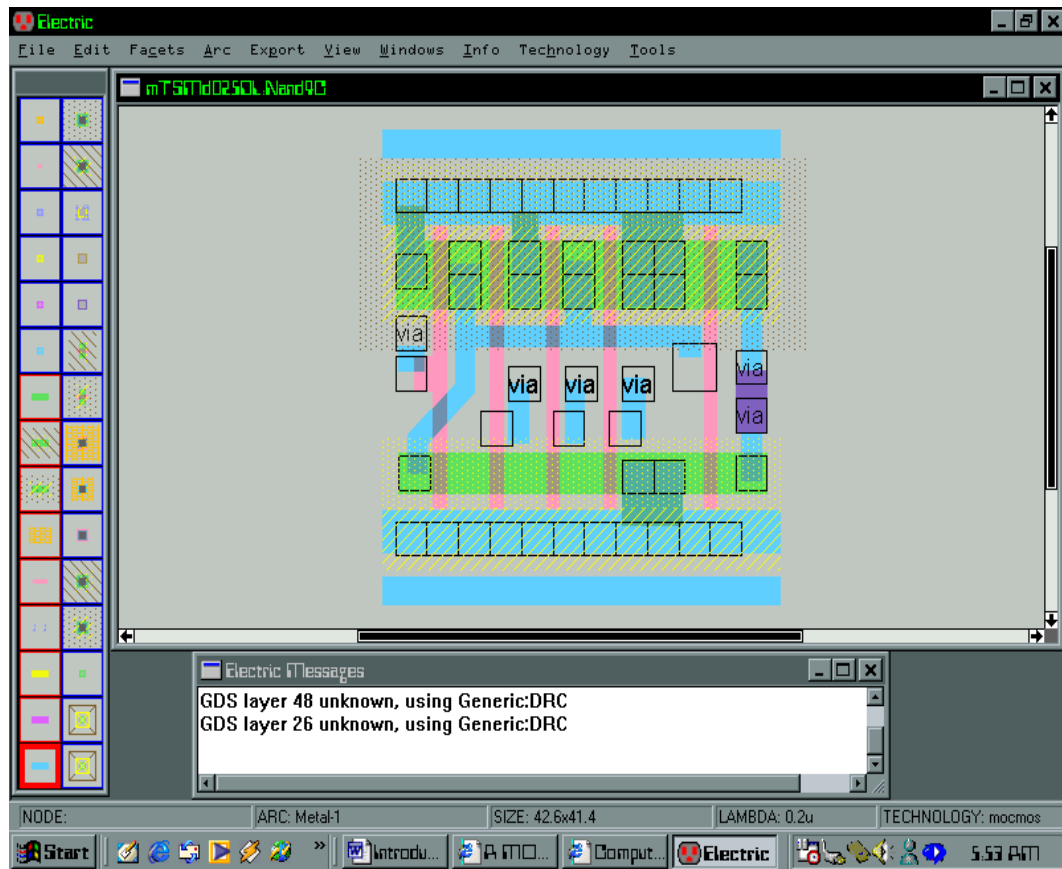


Figure 5.8: Electric interface

While setting up a CAD tool for a specific technology, the following has to be ensured for accurate functioning.

- ?? Verify layer mapping.
- ?? Verify minimum width and spacing rules.
- ?? Verify extraction parameters (so layout can be extracted into SPICE netlists)

Table 5.3: Verification before starting a layout

We can make maximum use of Electric, as it is open sourced and manually define the DRC rules and layer mapping. Once Electric is setup for a technology, layouts can be generated in either CIF or GDS II. As a final check, the layout generated by Electric can be imported into Cadence for testing. If the layout passes the DRC and LVS, we have a reliable design flow.

5.8 SUBMITTING TO MOSIS

MOSIS accepts designs in the CIF or GDS II format. It also requires a written indication of an official commitment of funds in the form of a purchase order or typed statement on corporate or institutional letterhead, signed by an

authorized purchasing agent. An optional design rule check is available to customers who wish to purchase this service. Designs may be submitted to MOSIS electronically by a web form, e-mail, FTP, or they may be delivered to MOSIS on physical media. The pricing information is available online at www.mosis.org

5.9 LAYOUT USING ELECTRIC

Electric™ is one of the most promising of all the CAD tool suites encountered during the making of this document. Table 5.4 shows the salient points of Electric

Salient features of Electric

1. *One CAD system, with a single user interface, can be used for both IC layout and schematics.*
2. *Open sourced software. source code available for download (www.gnu.org).*
3. *Source code manipulation and redistribution allowed.*
4. *Operating system independent.*
5. *Excellent documentation*
6. *Ideal tool for university environment.*
7. *Highly flexible and versatile.*
8. *Supports various import and export formats.*
9. *Industry compatible.*

Table 5.4: Electric salient feature

A typical chip layout system allows a designer to manipulate "paint", that is, he can edit polygons on different layers. When you cross polysilicon with active, it creates a MOS transistor. When one of these two pieces of paint are accidentally moved, the transistor ceases to exist. These systems don't understand the circuit in any sophisticated way; they just help the designer to manipulate the layout. Electric is different from these chip layout systems, as it works with actual components, rather than simple polygons. The MOS transistor is a single component that contains two layers: polysilicon and active. If the component is moved, both layers (and the connected wires) move. Like paint systems, Electric displays the full geometry on the screen. Unlike paint systems, Electric also understands the connectivity of the circuit.

5.10 LOADING VENDOR TECHNOLOGIES INTO ELECTRIC

Vendor technologies can be loaded into Electric by importing standard LEF libraries. Figure 4.6.1 shows the SCMOS_SUBM at lambda=0.30 ([AMI 0.80](#)) GDS II library loaded into Electric (Note: These are the free SCMOS libraries from MOSIS and only GDS II files are available and LEF libraries need to be bought). A D-flip flop layout is on the display in figure 5.9. Figure 5.10 shows the same

with SCMOS_SUBM at lambda=0.30 ([AMI 0.50](#)) GDS II CIF library loaded and all the standard cells expanded.

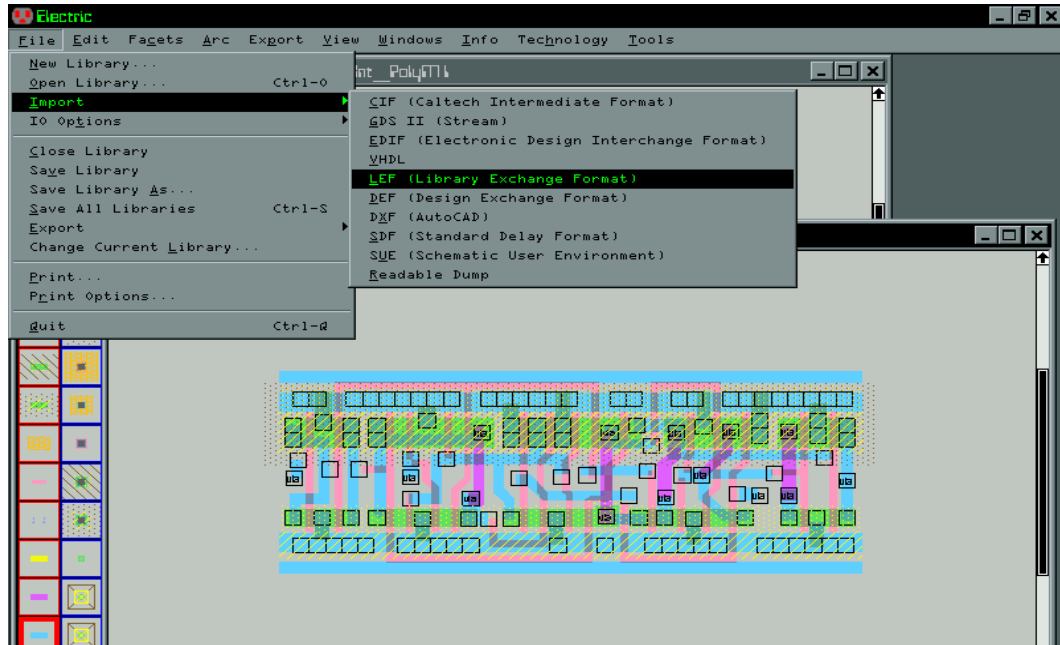


Figure5.9: SCMOS_SUBM at lambda=0.30 (Electric interface for importing LEF files)

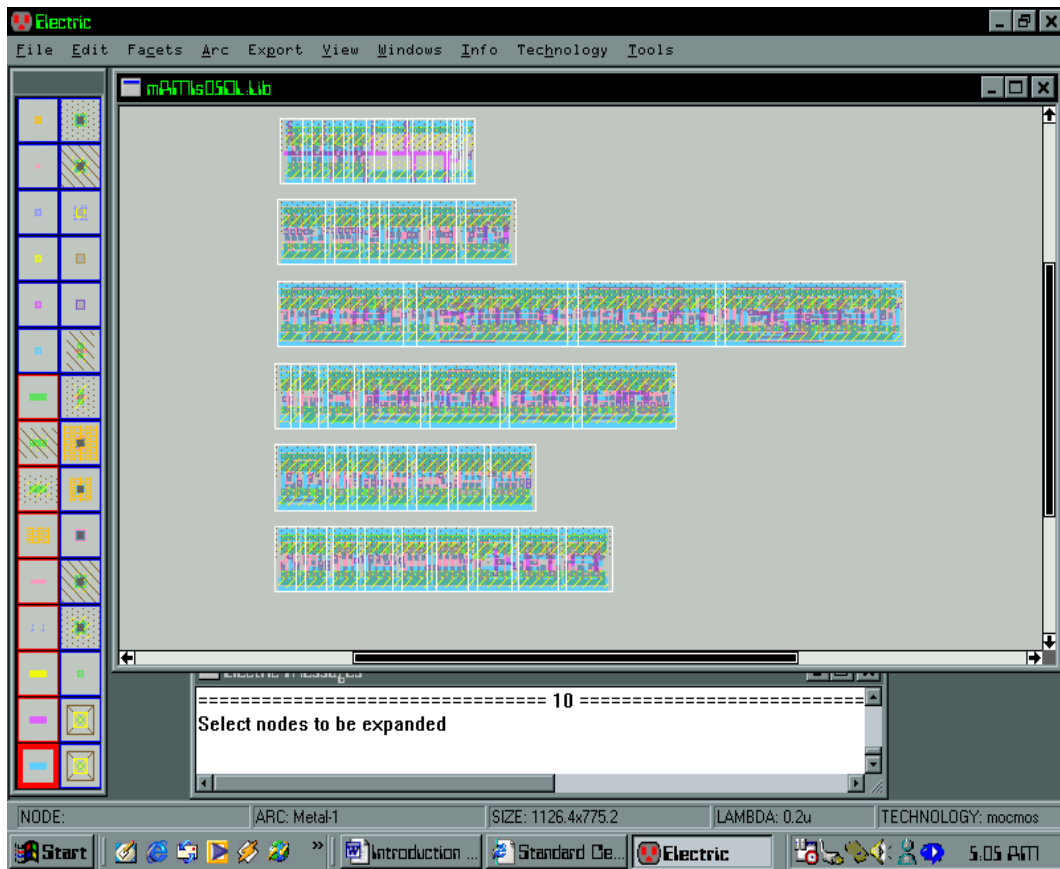


Figure 5.10: All cells of SCMOS_SUBM at lambda=0.30 (AMI 0.80)

5.11 LAYER MAPPING IN ELECTRIC

The GDS II / CIF files provided by the vendors have the complete layout description in terms of layers. Layers are usually numbers or identifiers associated with different materials that go into the making of a layout. Most of the times,

layer mapping of the vendor and the CAD tool do not match. Electric is highly configurable in this aspect and all the layers can be mapped according to the vendor's specification. Figure 5.11 shows the layer-mapping interface of Electric.

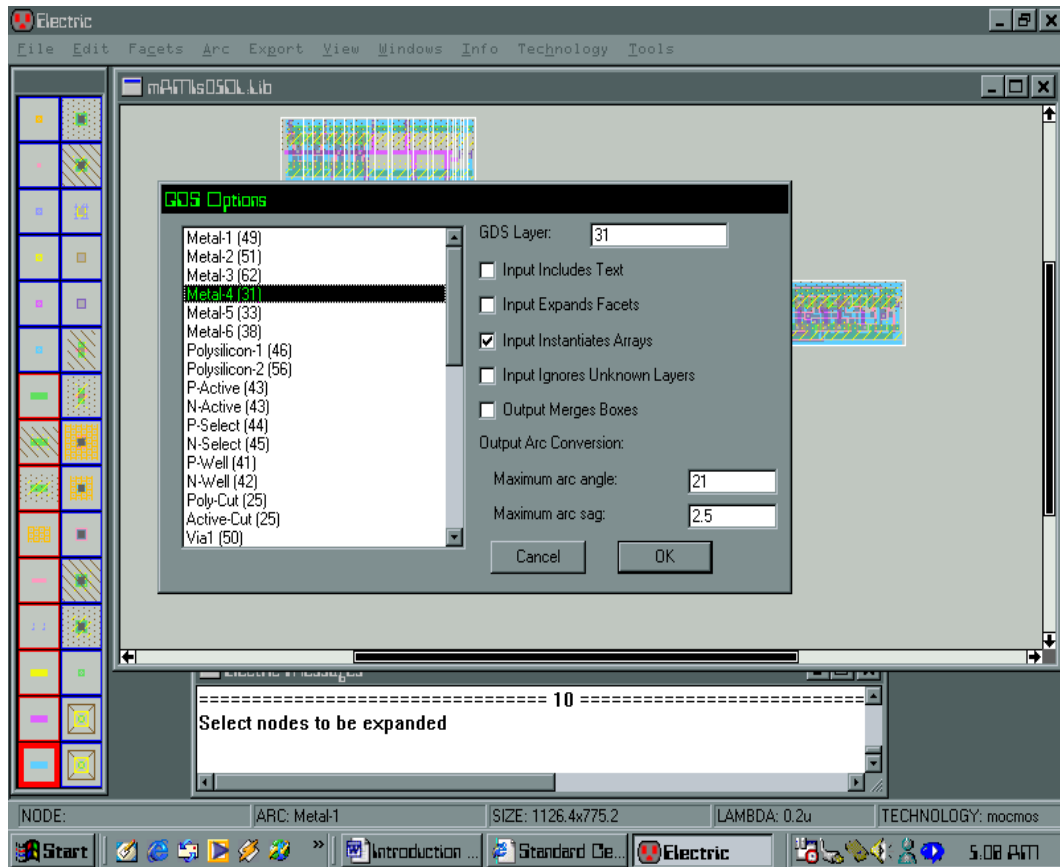


Figure 5.11: Flexible layer mapping interface in Electric

5.12 STANDARD CELL LAYOUT IN ELECTRIC

Layouts can be generated automatically or manually in Electric. Figure 5.12 shows the two processes.

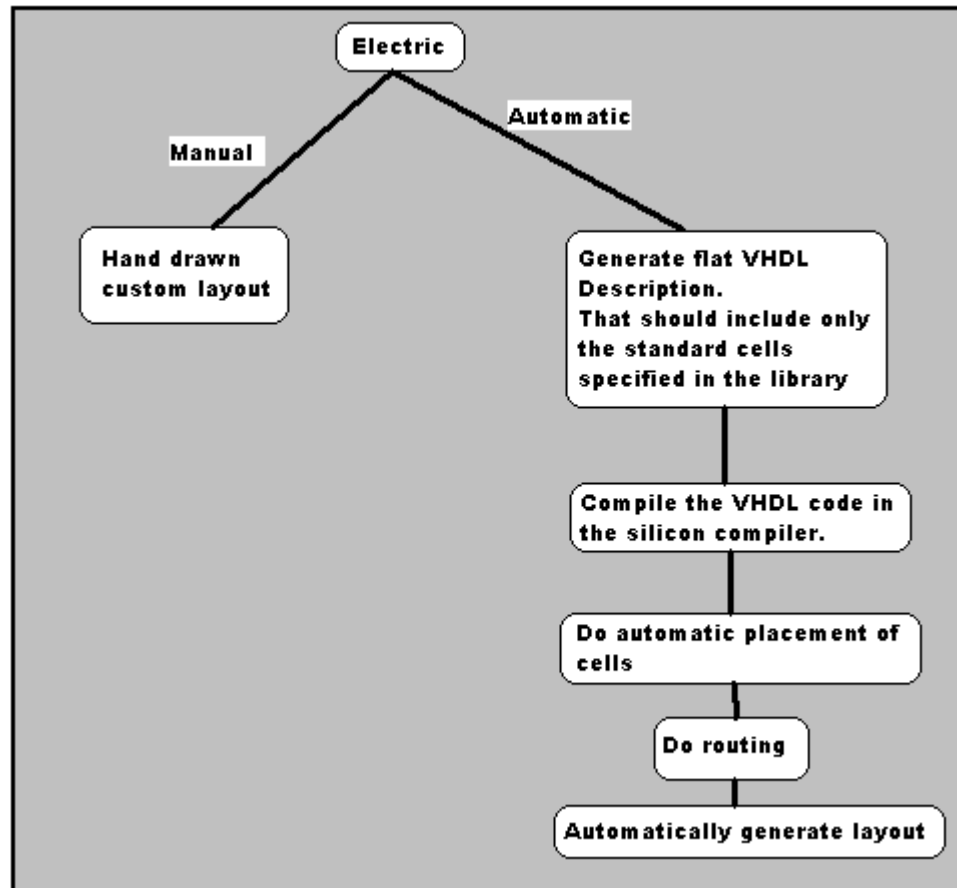


Figure 5.12: Standard cell layout formation in Electric

5.13 MANUAL LAYOUT GENERATION IN ELECTRIC

By choosing the appropriate layer and design materials from the 'Materials' menu, custom layouts can be drawn. Figure 5.13 shows the layout of a hypothetical circuit drawn in this fashion. This method is primarily used to generate cells and circuits that are unique and have custom defined behavior. The approach consumes a lot of time if large circuits are to be built from the transistor level. A useful feature in Electric allows these customized cells to be included in the standard cell library and reused in the Silicon Compiler (Section 5.15).

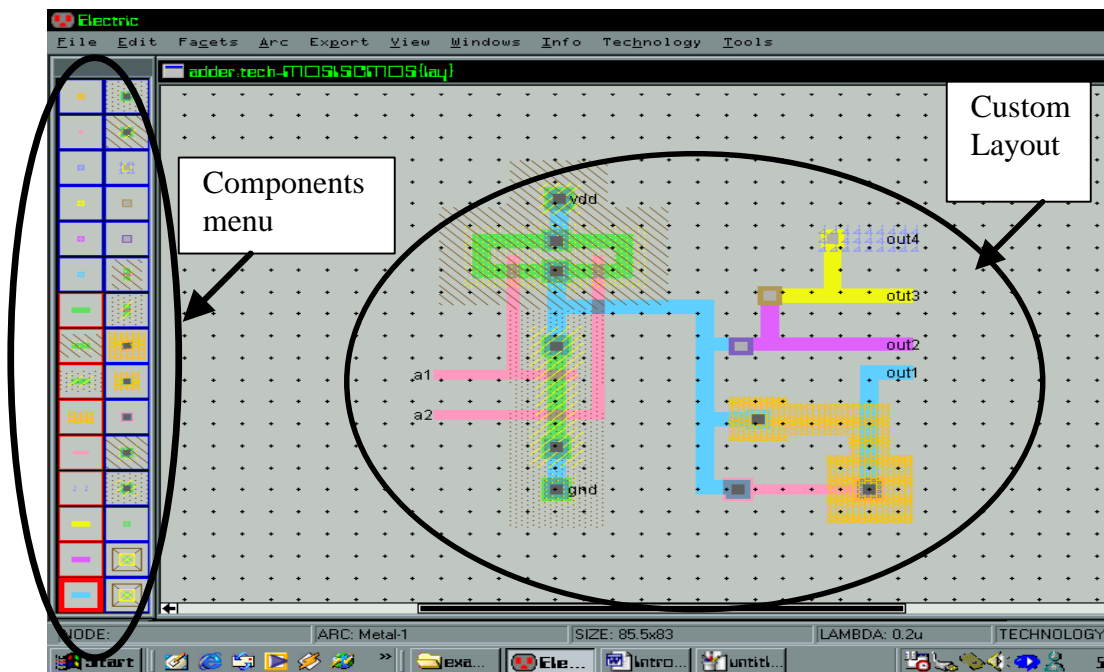


Figure 5.13: Manual Layout generation interface in Electric

5.14 MANUAL PLACEMENT AND SEMI-AUTOMATIC ROUTING OPTIONS

After making the customized standard cells, it is necessary to create instantiations of these cells and interconnect them to make the entire circuit. The process of interconnecting various cells is known as ‘routing’. Electric has an option to manually ‘place’ the cell instantiations according to the designer’s floor plan (shown in figure 5.14). To route these cells, semi automatic procedures enumerated in table 5.5 can be used.

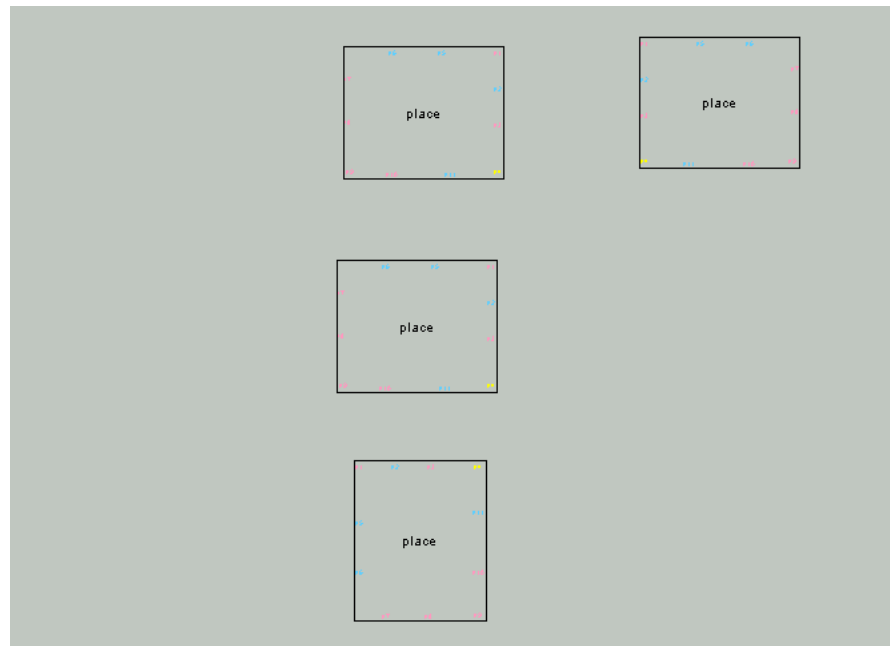


Figure 5.14: Designer’s floor plan

Semi automatic routing options

1. *Maze Routing*
2. *River Routing*
3. *Mimic Routing*

Table 5.5: semiautomatic routing options in Electric

The first step prior to initializing these routing engines is to interconnect the nodes using ‘Generic Arcs’. Generic arcs are standard wires and do not have any layer associated with them. They are a graphical method of defining the node interconnections. Figure 5.15 illustrates this procedure. The next step is initiating the routing engines. Figures 5.16, 5.17, 5.18 and 5.19 illustrate the results of maze and river routing routing .

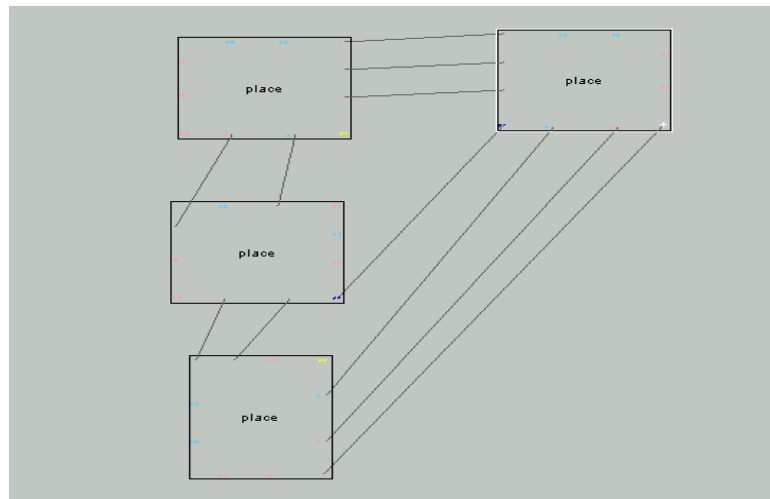


Figure 5.15: standard cells connected with generic arcs

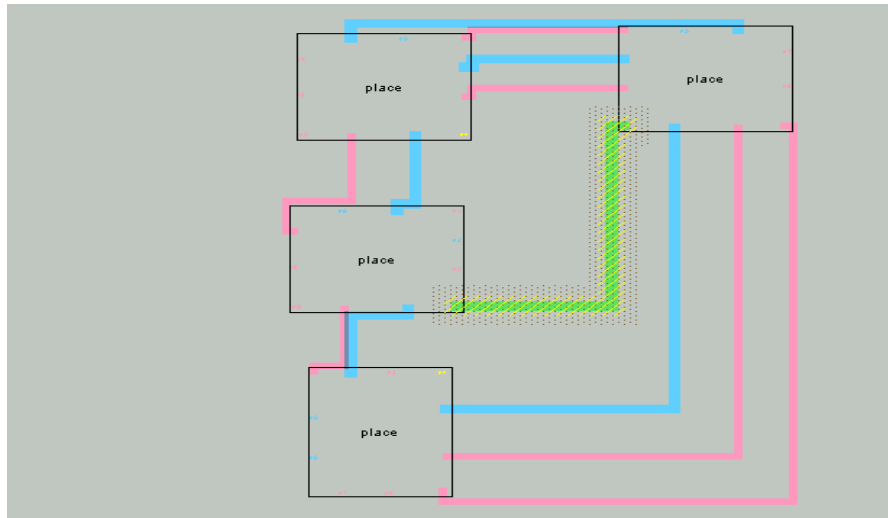


Figure 5.16: Results of maze routing

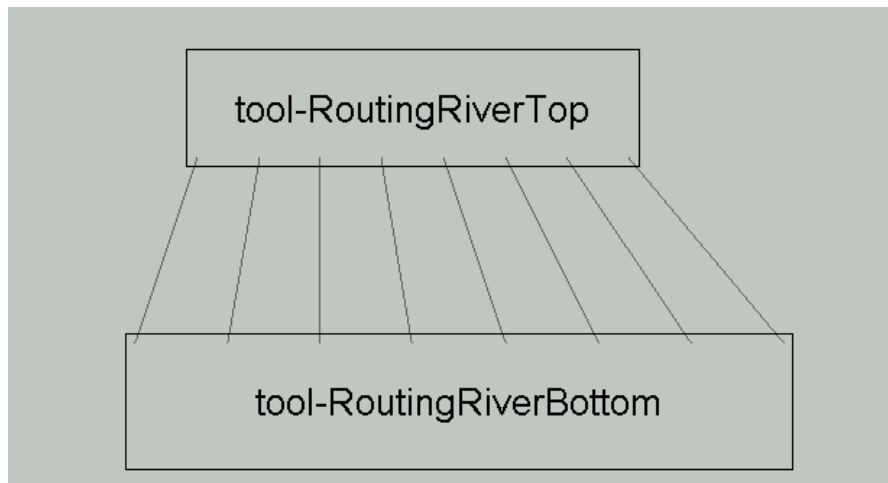


Figure 5.17: Generic arcs defining buses.

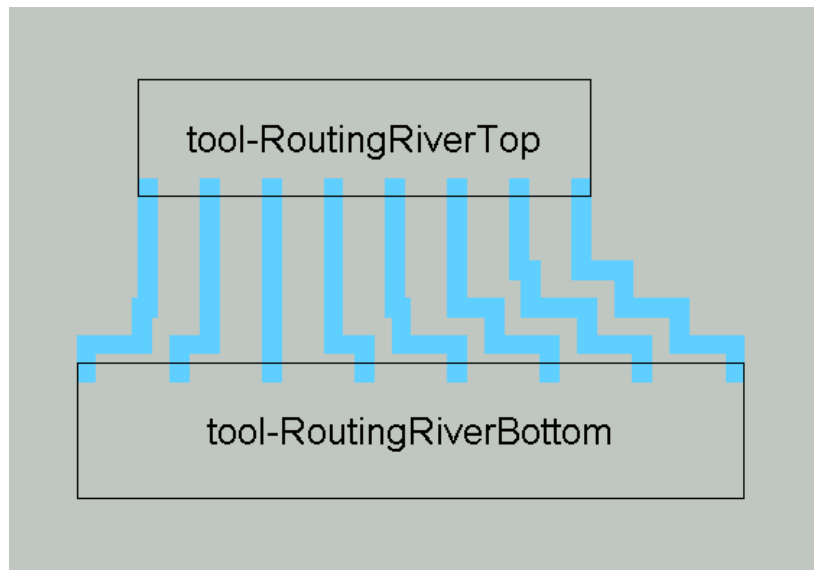


Figure 5.18: result of river routing buses

Some of the problems encountered with the routing engines are enumerated below in table 5.6.

1. *Routing can take place only one layer at a time.*
2. *Interconnecting nodes must be similar.*
3. *Topologically complex circuits cannot be routed.*
4. *Possible wastage of floor space.*
5. *Unprecedented parasitic elements.*

Table 5.6: Problems associated with semiautomatic routing

5.15 VHDL SYNTHESIS: SILICON COMPILER IN ELECTRIC

Vendors supply a variety of standard cells whose instances can be used in making a large layout. These cells are usually optimized for speed and space. There are expensive tools like Synopsis in the market that can read a VHDL/Verilog description, automatically 'place' these cells in a layout and run connections among them to form a layout.

Electric has such a synthesizer called the Silicon Compiler. The first step in creating an automatic layout is to create a flat VHDL description (A description that contains only instantiations of standard cells). The example chosen to illustrate the process of automatic layout generation is that of an 8-bit comparator, which is a fairly large, but a simple circuit. Table 5.7 shows the Flat VHDL code. Figure 5.19 shows a screen capture of the VHDL Code interface in Electric.

```

--ENTITY COMPARATOR_8
package buses is
type BUS8 is array (0 to 7) of BIT;
end buses;
use buses.all;
entity COMPARATOR_8 is port(ina, inb:in BUS8; eql: out BIT);
end COMPARATOR_8;
architecture COMPARATOR_BODY of COMPARATOR_8 is
component xor2 port(a1,a2: in BIT; y,yb: out BIT);
end component;
component nor2 port(a1,a2: in BIT; y: out BIT);
end component;
component nand4 port(a1,a2,a3,a4: in BIT; y : out BIT);
end component;
signal comp: BUS8;
signal p,q: BIT;
begin
XORG: for i in 0 to 7 generate
XORX: xor2 port map(ina(i),inb(i),open,comp(i));
end generate XORG;
NANDX1: nand4 port map(comp(0),comp(1),comp(2),comp(3),p);
NANDX2: nand4 port map(comp(4),comp(5),comp(6),comp(7),q);
NORX: nor2 port map(p,q,eql);
end COMPARATOR_BODY;

```

Table 5.7: Flat VHDL code for Silicon compiler

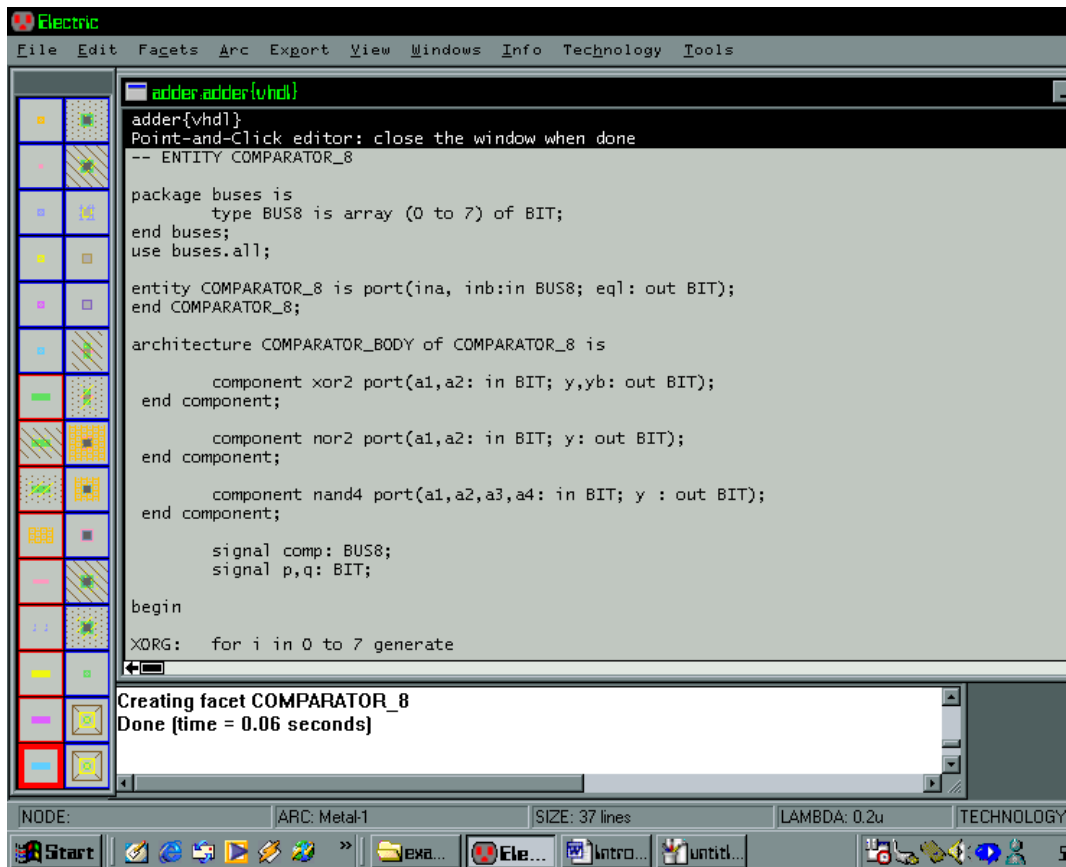


Figure 5.19: VHDL interface in Electric

The next step is to compile the VHDL code using the Silicon Compiler, place the standard cells and then route them. Figure 5.20 shows a screen

capture of this process. Figure 5.21 shows the final layout generated. Figure 5.22 shows the layout with all the standard unexpanded.

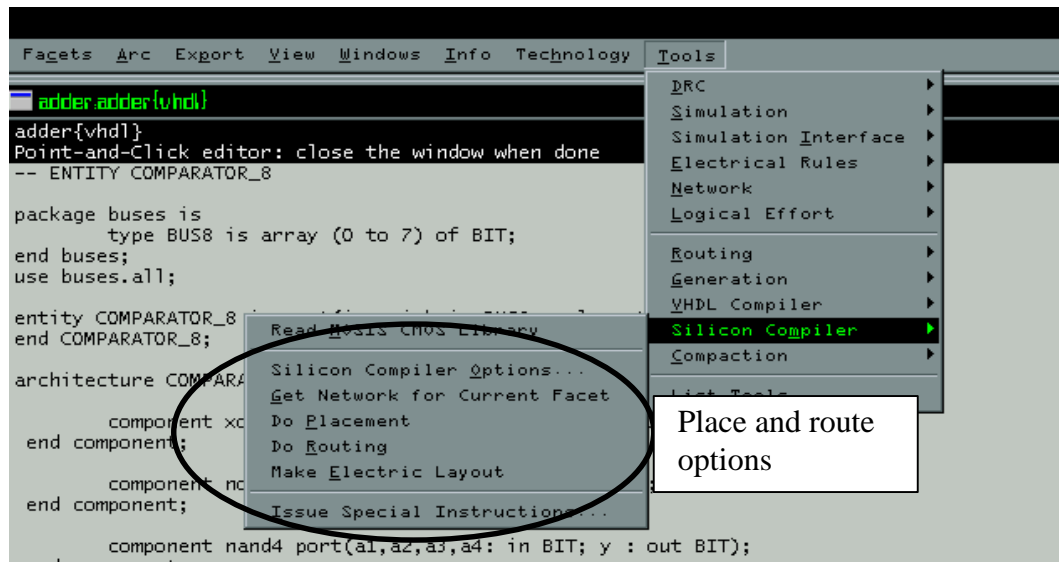


Figure 5.20: Place and route options

8 Bit comaprator layout

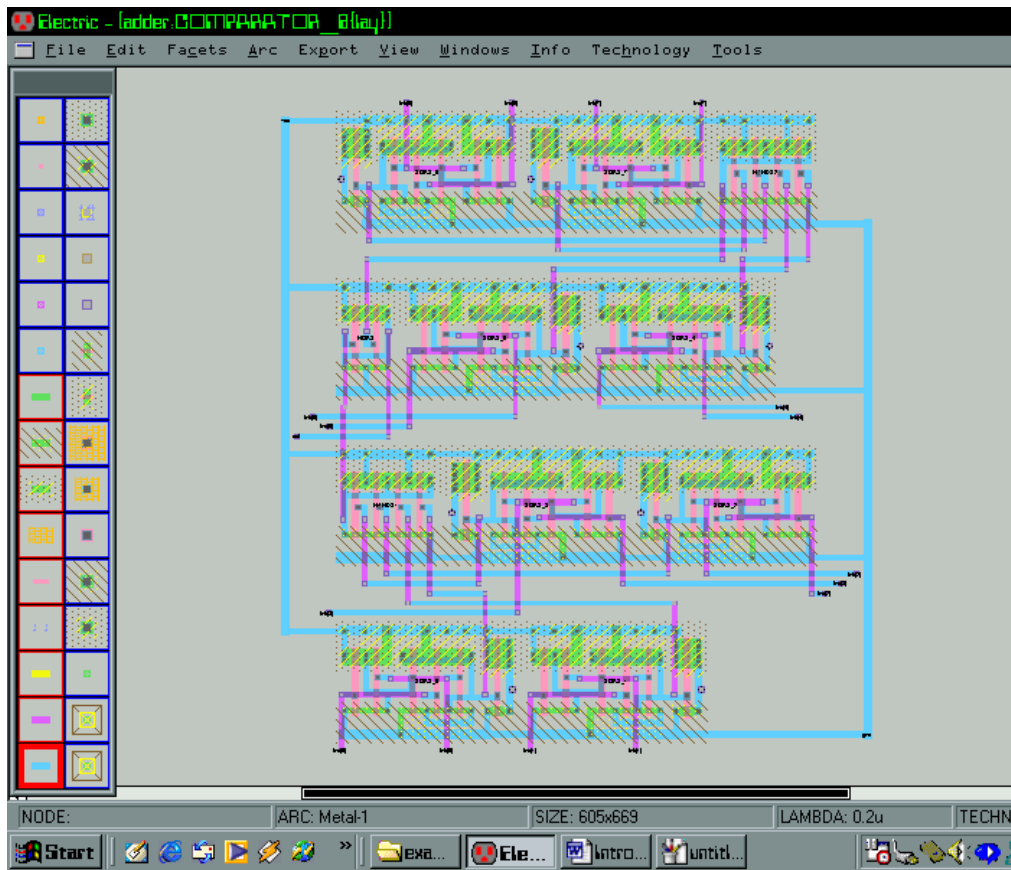


Figure 5.21: layout generated automatically

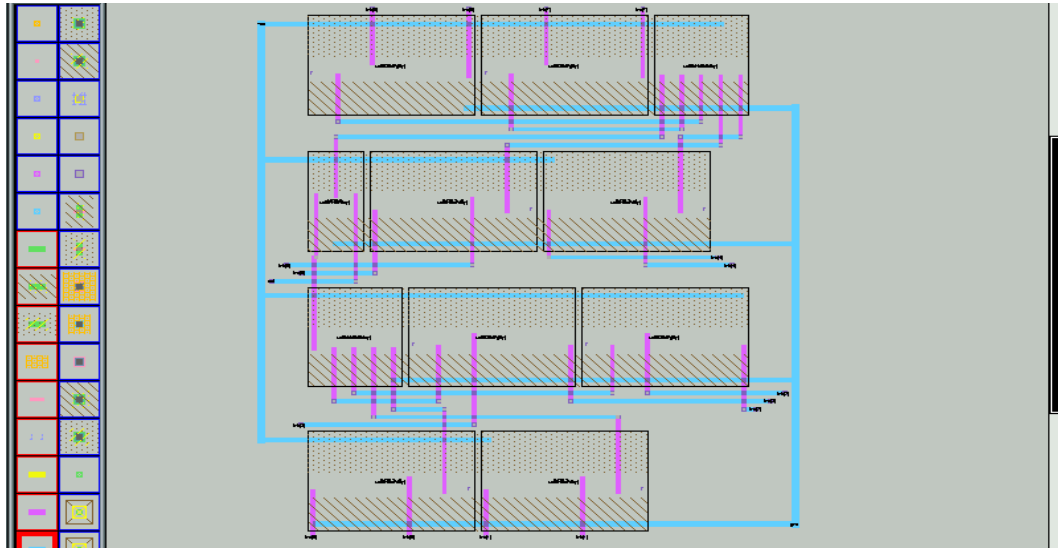


Figure 5.22: Standard cells used. Lines indicate the routed wires

The silicon compiler is highly flexible. Using the options available, the layout generation can be manipulated to make as close a fit to the original floor plan as possible. The options available are shown in Table 5.8.

1. *Horizontal routing : Metal 1 – Metal 6*
2. *Vertical routing : Metal 1 – Metal 6*
3. *Main power line : Horizontal/ Vertical.*
4. *Main Power line width.*
5. *Minimum distance between Cells.*
6. *Maximum number of horizontal Cells*

Table 5.8: Silicon compiler options

5.16 SIMULATION

Electric has both analog and digital simulators. The event based digital simulator is called ALS (Asynchronous Logic Simulator) and the analog simulator is SPICE. They are not very interactive but are accurate. Figure 5.24 shows the SPICE simulation results of the inverter shown in Figure 5.23. Figure 5.25 shows a digital simulation of the same.

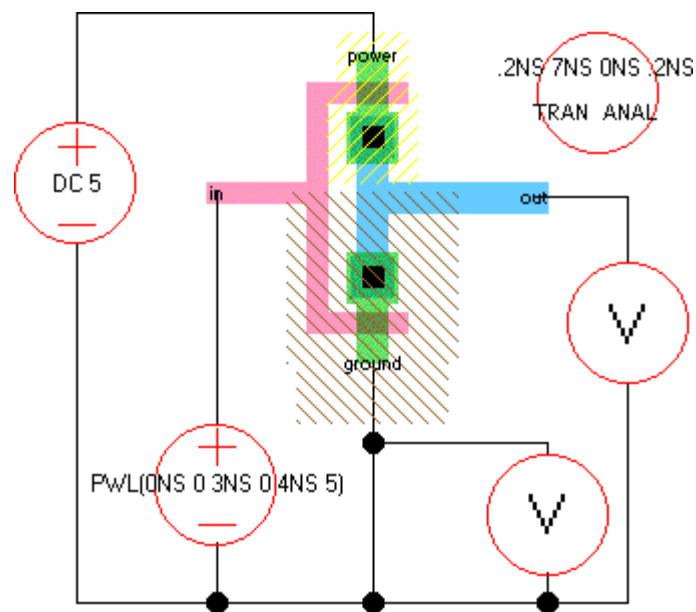


Figure 5.23: Inverter layout under simulation

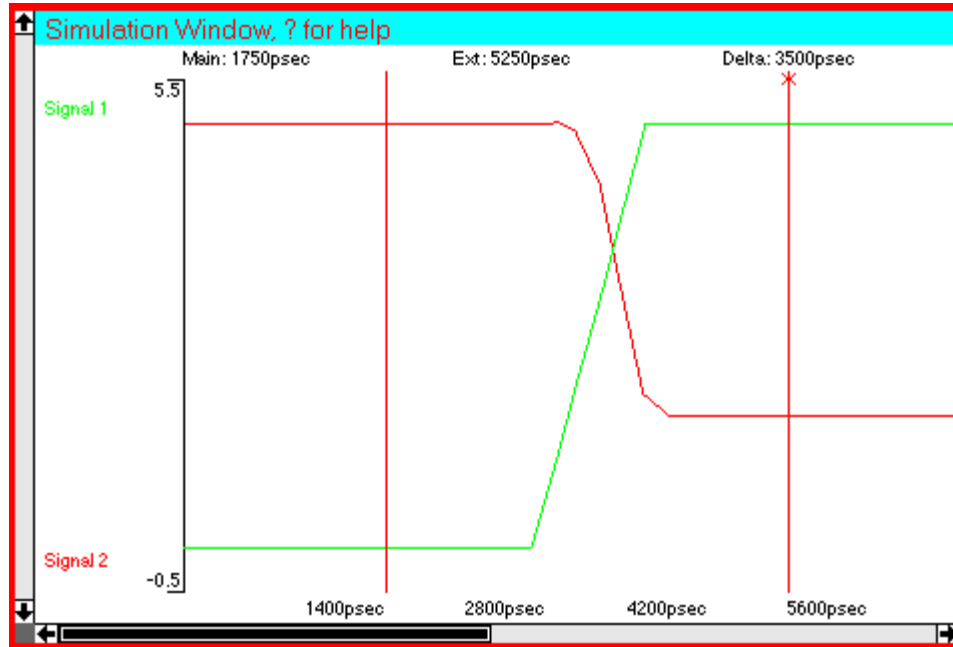


Figure 5.24: SPICE simulation

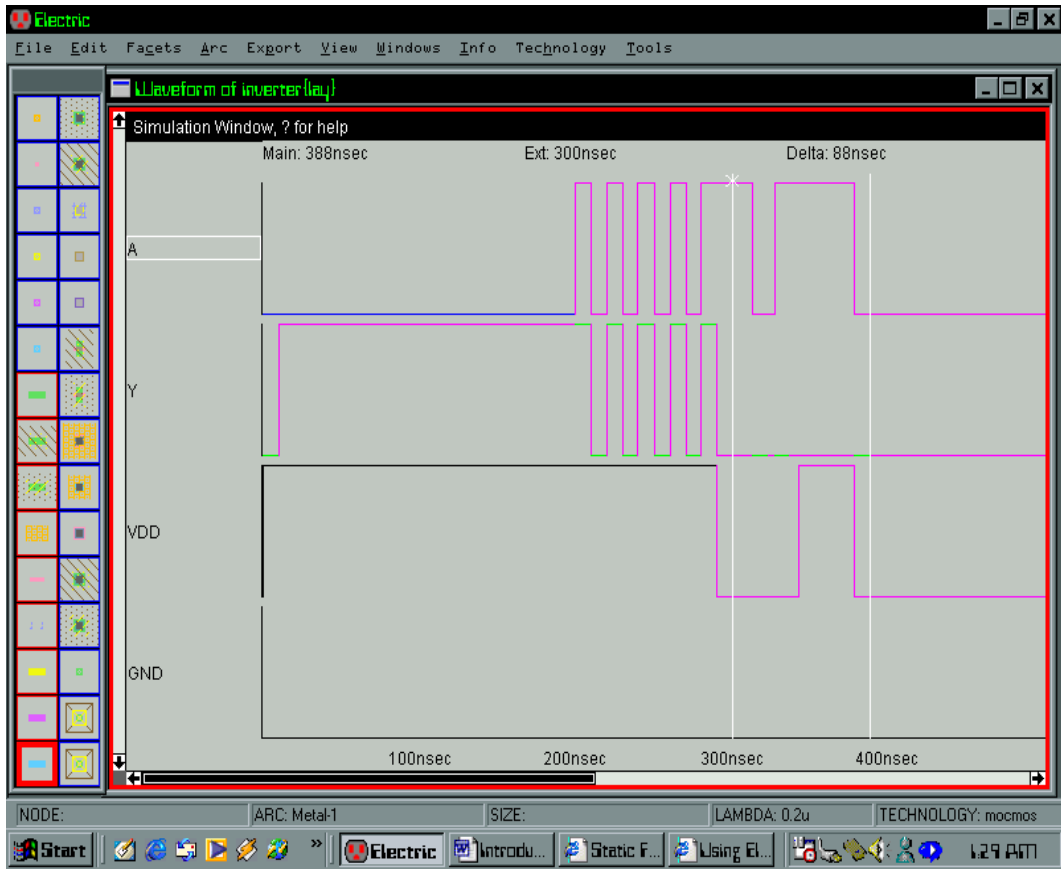


Figure 5.25: ALS simulation

Apart from these internal simulators, Electric can export netlist information in a variety of formats that can be read by other simulators. The only constraint being that, for accurate simulation results, these simulators must have the same vendor technology specifications loaded into them. Table 5.9 shows a list of Export Formats available in Electric.

Export Formats in Electric

1. *SPICE deck*
2. *Fast Henry*
3. *IRSIM*
4. *ESIM*
5. *RSIM*
6. *RNL*
7. *COSMOS*
8. *MOSSIM*
9. *TEGAS*
10. *SILOS*
11. *PAL*

Table 5.9: List of simulation export formats in Electric

5.17 MIXED SIGNAL DESIGN AND EXPORTING TO CADENCE

Mixed signal design involves two distinct phases design; the analog and the digital phase. They can be done simultaneously or one after the other. Though mixed signal design can be done in Electric, some analog designers might prefer to do their analog design in other CAD tool suites. One such tool could be Cadence. In mixed signal design, the bulk of design time is spent on the analog design, which is usually a couple of hundred transistors. The digital layout, consisting of probably

a hundred thousand gates, needs to be designed in short period of time. For a quick mixed signal digital design flow, this document proposes the following.

- ?? Do the analog part of design in Cadence.
- ?? Use Electric's Silicon Compiler for digital design to quickly generate CIF or GDS II layout files.
- ?? Export the layout in CIF or GDS II from Electric.
- ?? Import the CIF or GDS II file into Cadence as a single cell.
- ?? Verify DRC in Cadence.
- ?? Create a 'topcell' that contains instantiations of the analog cell and the digital cell
- ?? Do manual place and route. Note: This should not be very tedious as there are just two cells.
- ?? Verify DRC.
- ?? Extract CIF or GDS II file and ship to foundry.

Figure 5.26 shows a CIF file exported from Electric and Imported into Cadence. Electric's credibility can also be verified by performing a DRC in Cadence.

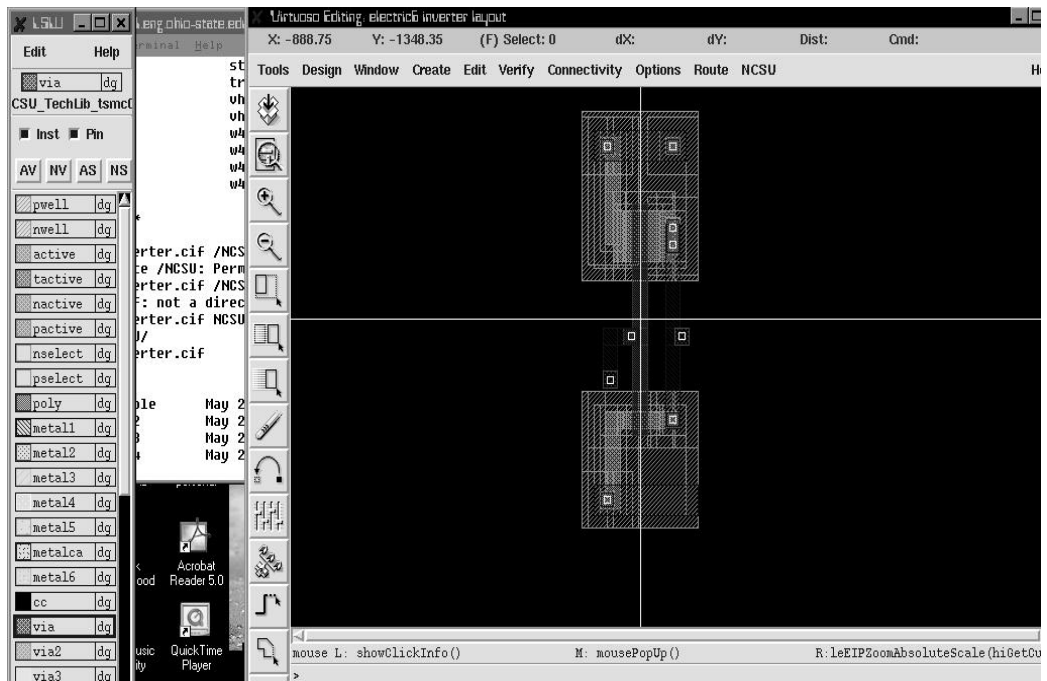


Figure 5.26: Inverter design imported into Cadence

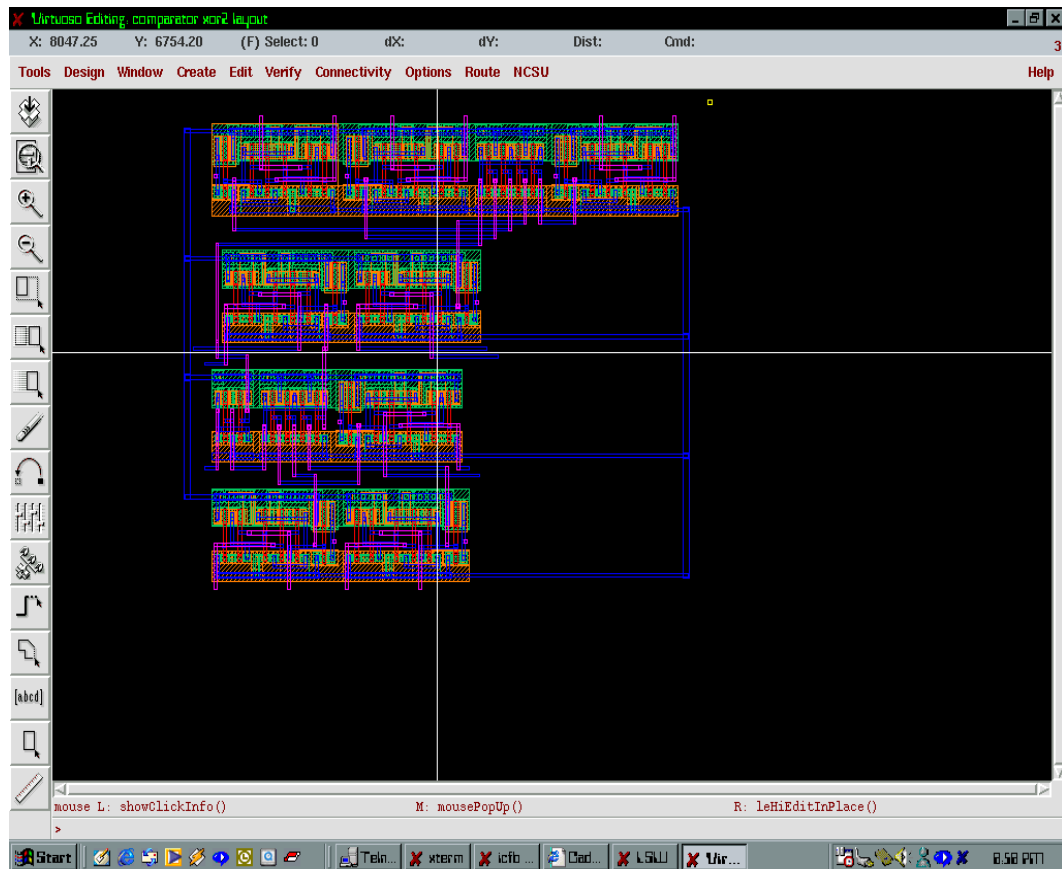


Figure 5.27: 8 bit comparator imported into Cadence Virtuoso

In the Ohio State University, SCMOS AMI 0.8 is absent in the Cadence suite. However, a library close to the process technology feature size being used can be used to verify the DRC (AMI 0.5 was used). Figure 5.28 shows the DRC results in Cadence. All the design rules are passed except the 'Via' sizes. This is expected, as libraries do not match. This experiment strengthens the reliability of electric as it has been proved that it can churn out industrially acceptable layouts.

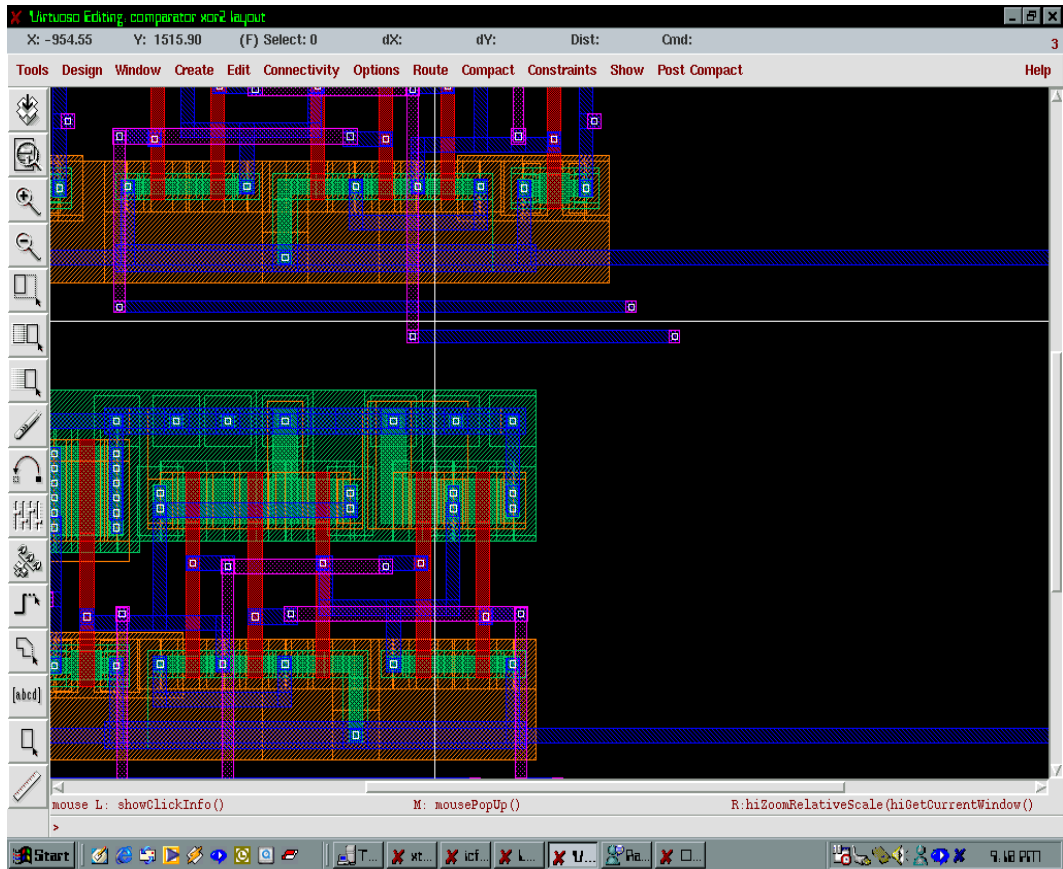


Figure 5.28: DRC failure due to mismatching Libraries

5.18 DRC

The DRC (Design Rule Check) is entirely dependent on the process technology. The design rules are provided by the vendor with the standard cell library. The LEF files provided by the vendor usually have this information.

Electric has a DRC option that automatically loads the design rules, provided the LEF files exist. These rules are flexible and can be customized. Figure 5.29 shows the DRC customization window.

There are three ways in which DRC can be evoked in Electric. Table 5.10 illustrates the DRC Options

DRC Options

1. *Hierarchical rule check: The design rules are checked from the bottommost cell and proceed in a hierarchical manner to the topmost cell that is instantiated.*
2. *Non-Hierarchical rule check: The layout is treated as a flat design and the rules are checked. This method is usually employed as a final check at the end of the design stage where all the cells are expanded.*
3. *Incremental Rule check: The design rules are checked at every stage when the layout is modified. This helps in custom layout design where instantaneous feedback is obtained regarding the legitimacy of the action.*

Table 5.10: DRC options

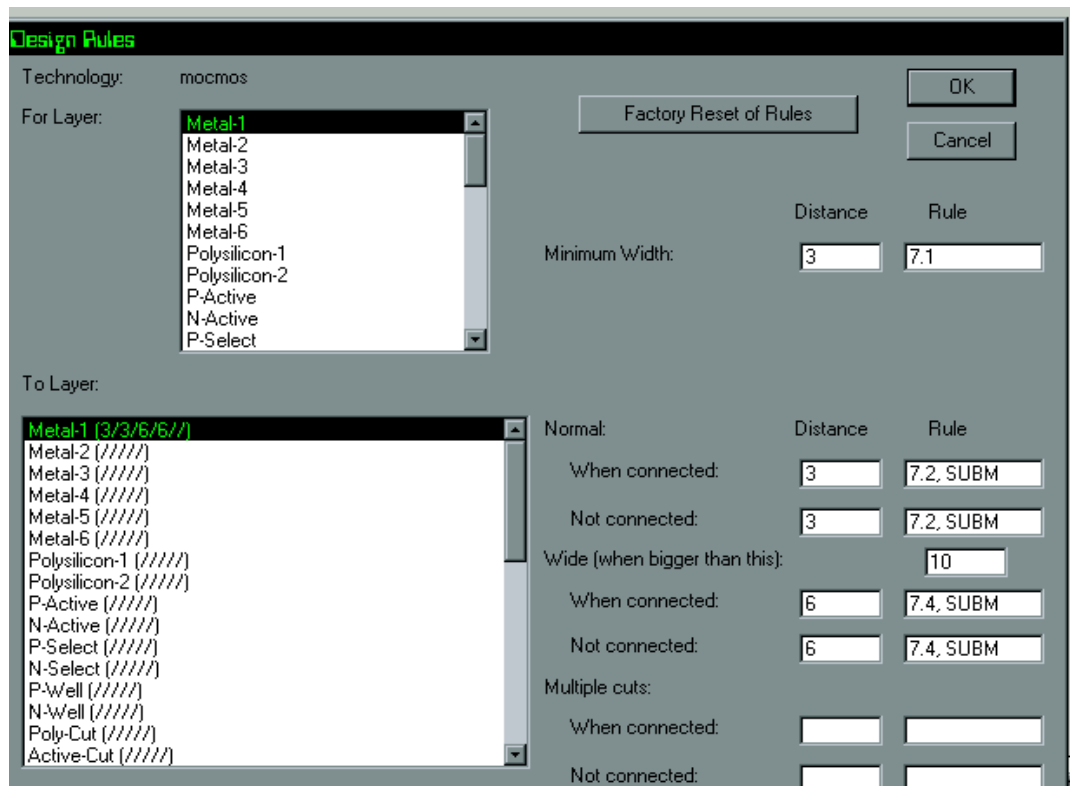


Figure 5.29: DRC customization window

5.19 LVS

Electric does not have an LVS (Layout Versus Schematic) option. This is one of the drawbacks of Electric. However, this feature is not very critical in Electric because the schematic can be extracted from the layout. This ensures that the LVS is implicitly satisfied. The other option is to import a schematic's EDIF

file and its corresponding GDS II/ CIF into Cadence and then run an LVS check. Figure 5.30 shows the schematic automatically extracted by Electric for the 8-bit comparator.

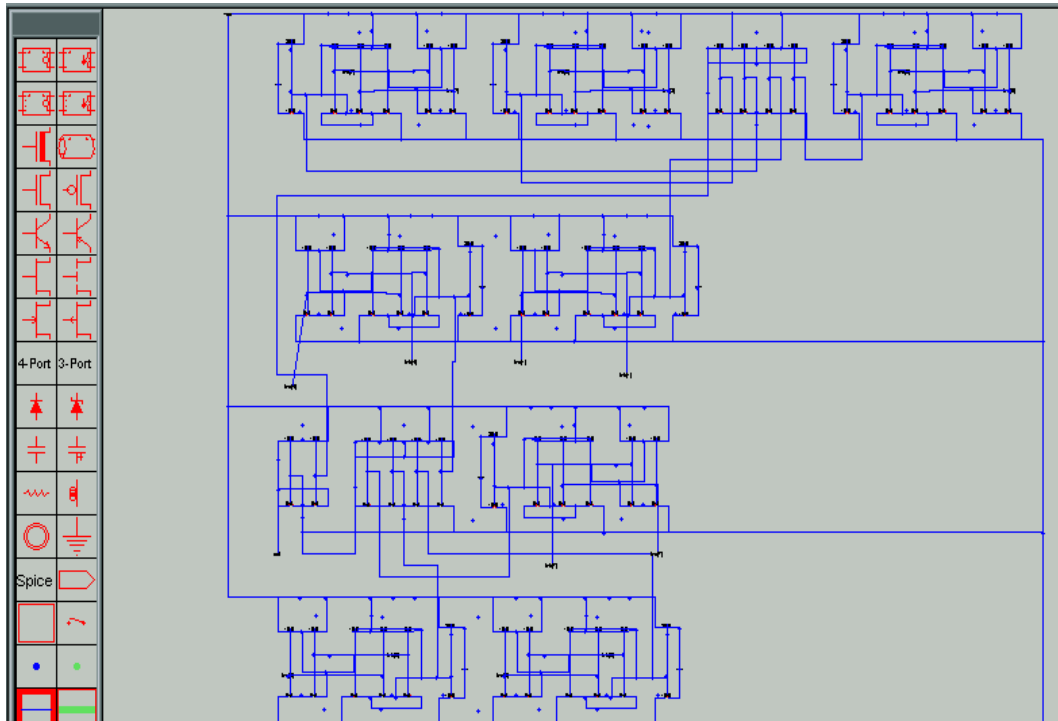


Figure 5.30: Automatic Schematic extraction in Electric

5.20 PAD FRAME

The final step before submitting the GDS II/ CIF to the foundry is to wrap the design in a Pad Frame. Pads define the connections that map to the pins of an IC. Pads need to be supplied by the vendor. Pads form a part of the vendor

supplied standard cell library. Figure 5.31 shows the different kinds of pads available in MOSIS TSMC 0.25 technology.

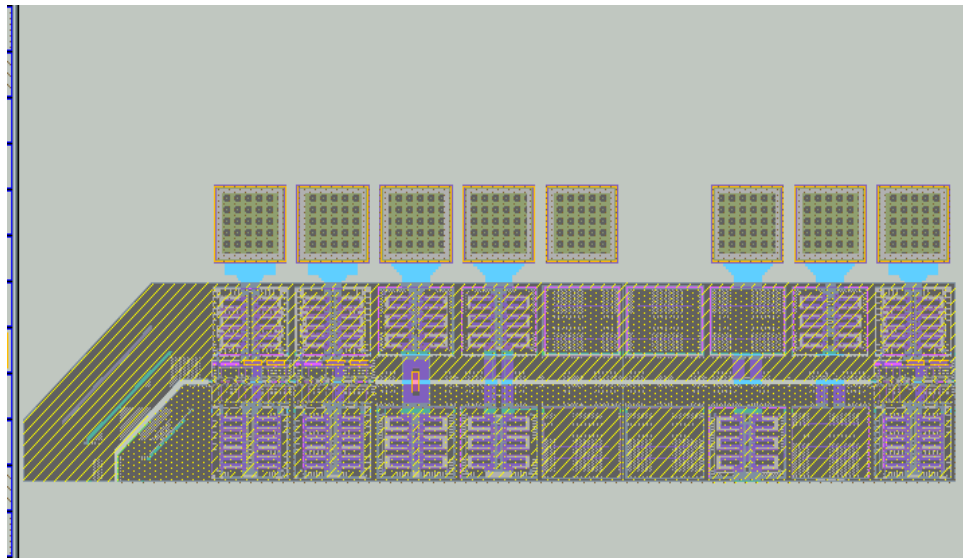


Figure 5.31: Different Pads available in MOSIS TSMC 0.25

Electric has an intelligent pad frame generation system. By reading the Pad GDS II / CIF files, it can automatically place the core and route wires to a pad frame. However, a script file of type ‘*.arr’ needs to be specified by the designer which has information about the kind of pads and the number of pads that need to be generated. Shown in Figure 5.32 shows the 8-bit comparator core in a pad

frame. Figure 5.33 shows a zoomed in view of a pad. Figure 5.34 shows the complete pad frame with the 8-bit comparator core. (*Note: The free library available had only 7 pins. Thus only 7 nodes are routed*).

This is the end of the design cycle for digital design. All that remains is to ship the GDS II / CIF files to a fabrication plant to get an IC.

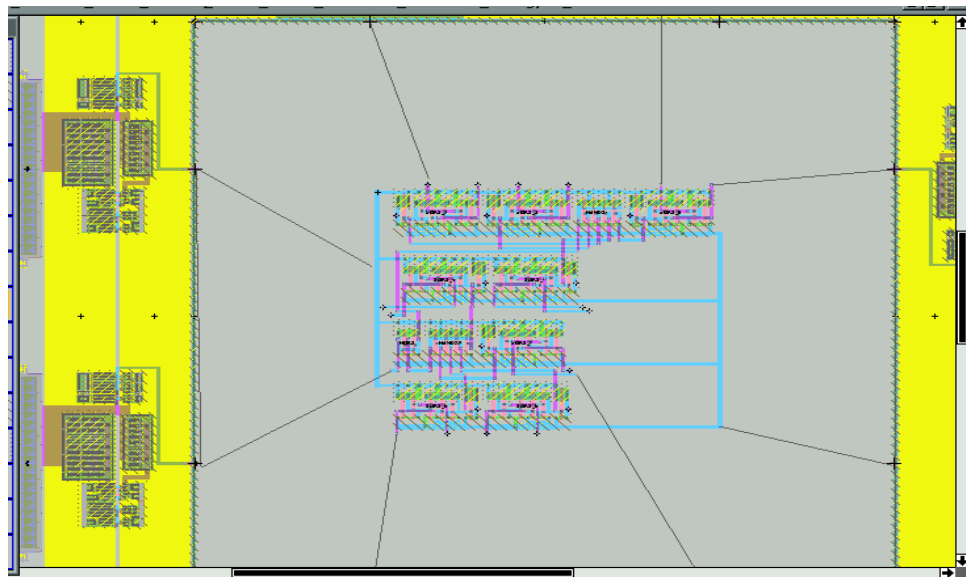


Figure 5.32: 8 Bit comparator core in Pad Frame

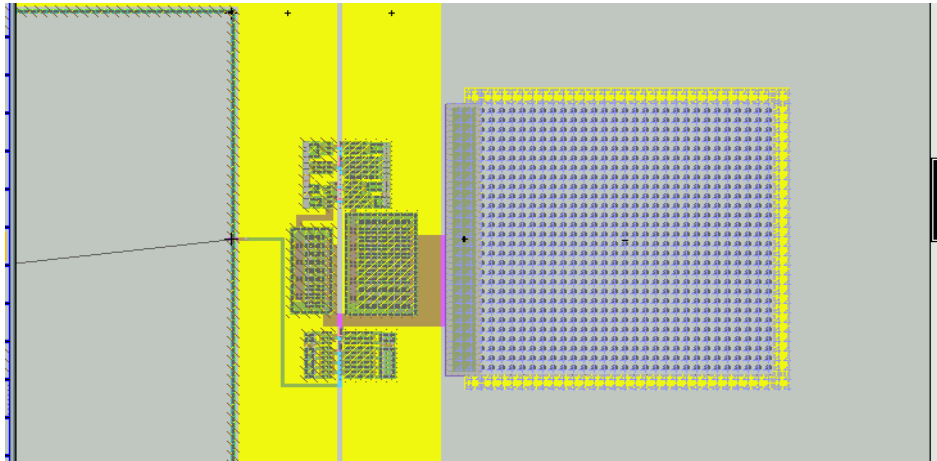


Figure 5.33: Zoomed in View of a PAD

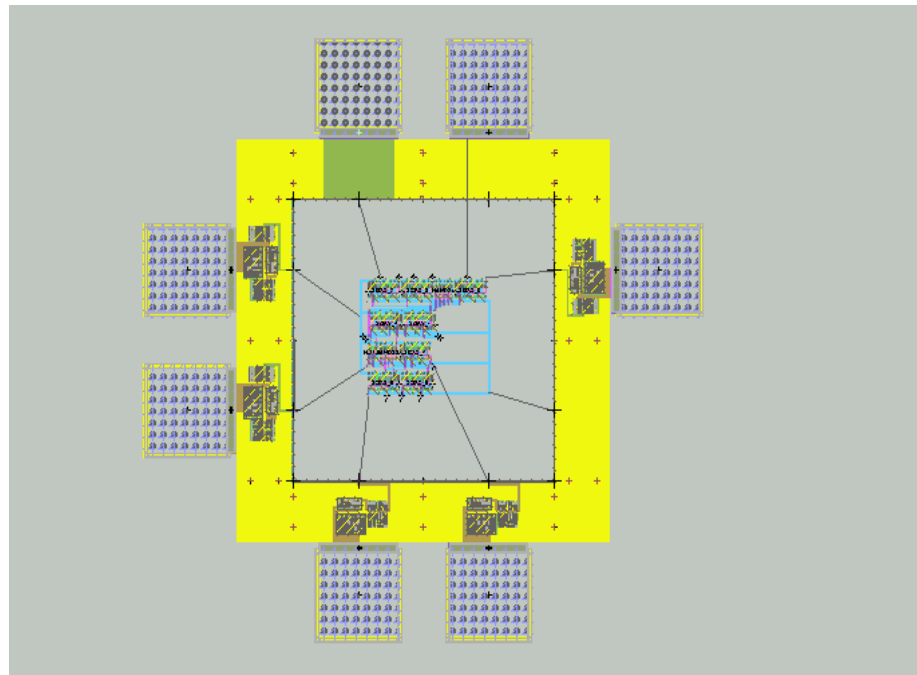


Figure 5.34: The complete chip

5.21 OTHER INTERESTING FEATURES IN ELECTRIC

3-D View:

Electric has other interesting features like a 3-D view of the layout. This could be a very handy feature in teaching. Figure 5.35 shows the Layout of a Dflip-flop. Figure 5.36 shows the same layout at another angle.

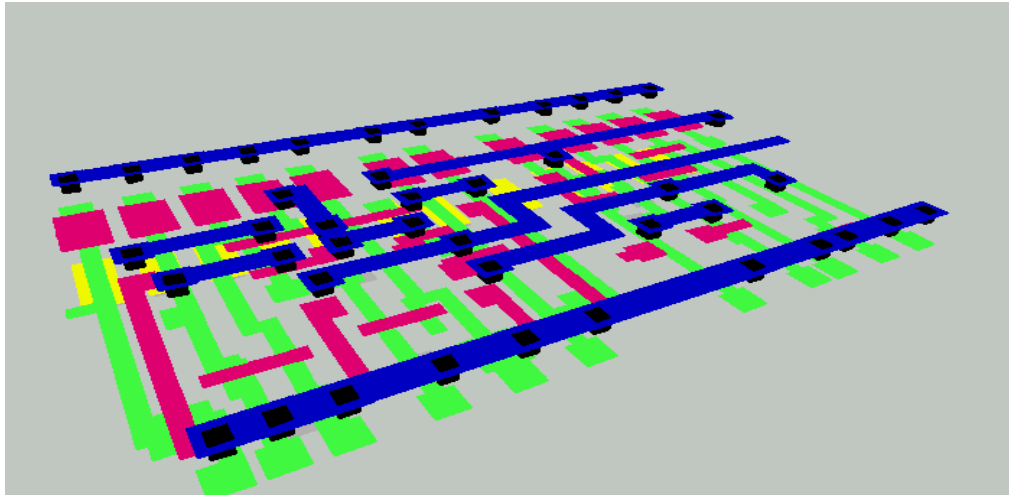


Figure 5.35: 3-D view of a Dflipflop

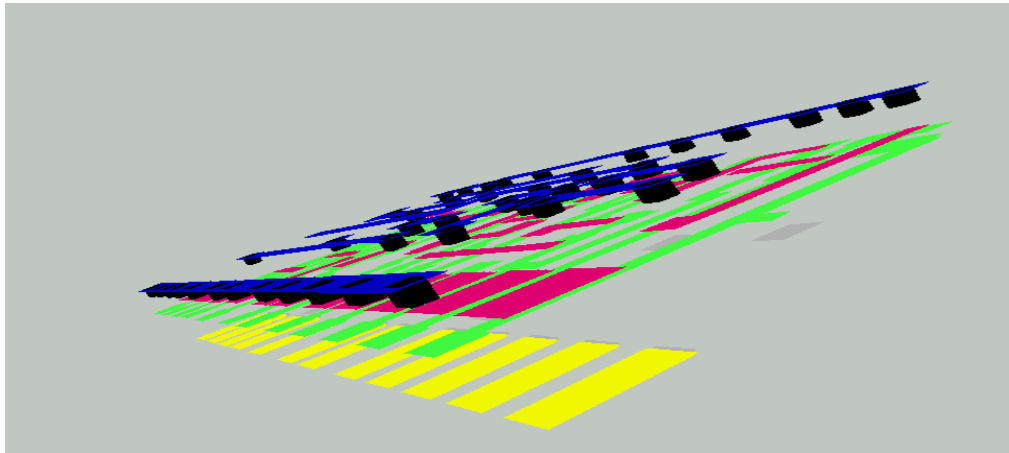


Figure 5.36: 3-D view of a Dflipflop

Compaction:

Compaction is a very useful tool when layouts need to be adjusted for minimum area without violating the design rules. Compaction does not change the entire geometry of the layout. The compaction tool squeezes layout down to minimal design-rule spacing. It does this by alternating horizontal and vertical directions of compaction until no further space can be found without violating the design rules. Figure 5.37 shows a wasteful inverter layout. Figure 5.38 shows the layout after compaction

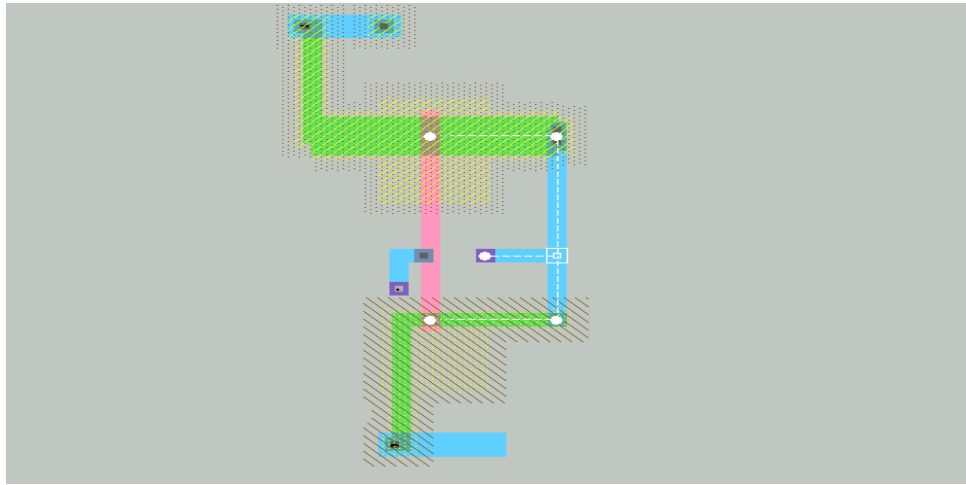


Figure 5.37: Wasteful layout of an inverter

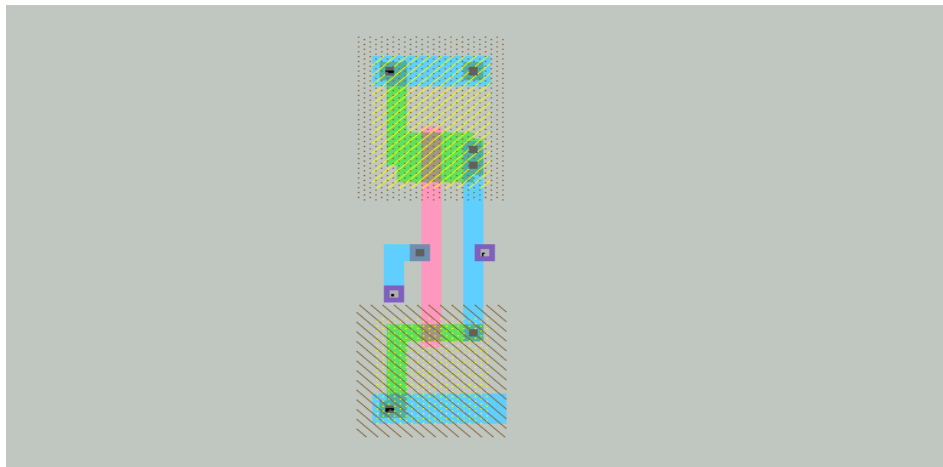


Figure 5.38: Layout after compaction

Logical Effort:

The Logical Effort tool examines a digital schematic and determines the optimal transistor size to be used in order to get optimized speed. When the Logical Effort tool runs, it annotates each digital schematic gate with a fanout ratio that can be used to size the transistors inside of that gate. It is up to the designer to use this information in the actual IC layout. This tool can be very useful when trying to optimize critical path delays. Figure 4.14.5 shows an arbitrary circuit that has been analyzed by Logical Effort. 'C' shows the capacitances at each node and 'h' shows the optimized fanout required for that gate. By using this information, the designer can layout his transistors according to the 'h' value. For example, if 'h' is equal to 1.8, the designer needs to layout a transistor that needs to drive approximately 2 gates. If the transistor is made too large, it would drive the succeeding gates much faster but it would load the previous stage. Hence the previous stage becomes slower. If the transistor were made smaller, it would drive the next stage much slower. It is thus left to the discretion of the designer to size the gates according to the value 'h'. A good approximate solution for 'h'=1.8 would be a transistor of width 2?.

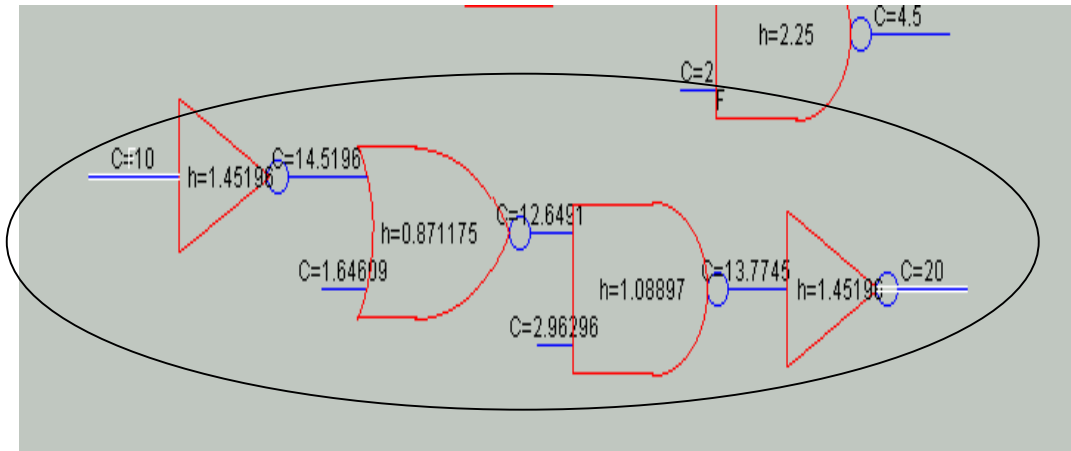


Figure 5.39: Logical Effort results.

PLA Generation:

Electric's PLA generator tool consists of two different generators: an nMOS generator and a CMOS generator. Both use personality tables to specify which taps in the programming array are set. Both produce a hierarchical array specification made up of AND tables, OR tables, drivers, and all necessary power and ground wires. Shown in figure 5.40 is the CMOS PLA generated for the function given in table 5.11.

$$f = (a \text{ and } b \text{ and } (\text{not } c)) \text{ or } ((\text{not } b) \text{ and } (\text{not } a))$$

$$g = (a \text{ and } c) \text{ or } ((\text{not } a) \text{ and } (\text{not } c))$$

Table 5.11: PLA generation equations

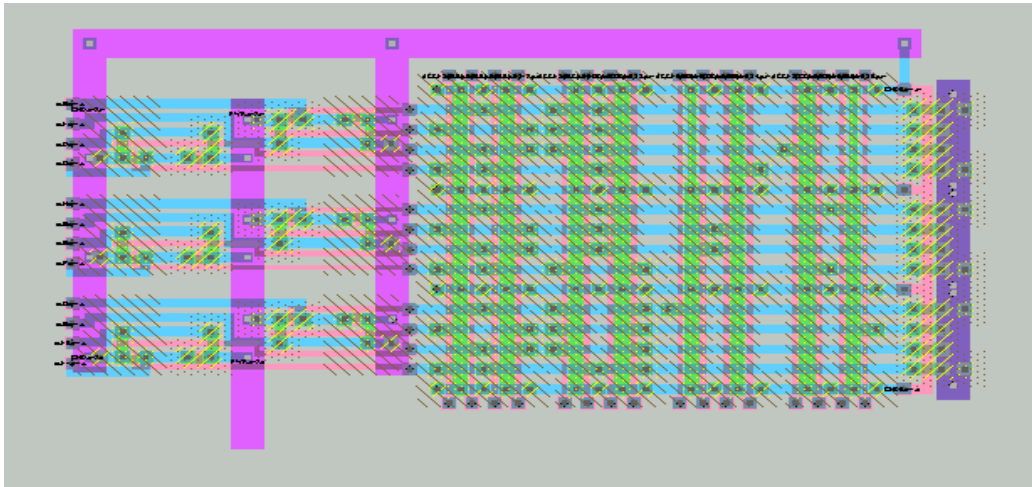


Figure 5.40: PLA generated for equations in table 5.11

CHAPTER 6

SUMMARY AND CONCLUSION

6.1 SUMMARY

The aim of this thesis was to find an alternative digital design flow. An attempt was made to achieve this with primarily 3 tools.

?? TestBencher

?? X-HDL3

?? Electric

Table 6.1 highlights the Salient points of each of these tools

TestBencher salient points :

1. *Used for Design entry*
2. *Used for Behavioral description*
3. *Used for structural description*
4. *Saves time with automatic test bench generation.*
5. *Can be used for post silicon testing.*
6. *Supported on all Operating systems.*
7. *Priced at \$1200.00*

X-HDL3 Salient points

1. *Used for Verilog to VHDL conversion*
2. *Used for VHDL to Verilog conversion*
3. *Used for generating synthesizable HDL code.*
4. *Supported on Windows 95,2000,NT, UNIX, LINUX.*
5. *Priced at \$800.*

Electric Salient points

1. *Used for layout generation (manual, automatic).*
2. *Place and route.*
3. *Industry compatible.*
4. *Source code available for download*
5. *Supported on all OS platforms*
6. *Priced at \$0.00*

Table 6.1: Salient features of TestBencher, X-HDL3, Electric

The thrust of this thesis has been to find a digital design flow that is most agreeable in a university based environment. Figures 6.1 and 6.2 show some of the recommended digital design flows for the Ohio State University. The designer can choose any of the other options shown in the Figures.

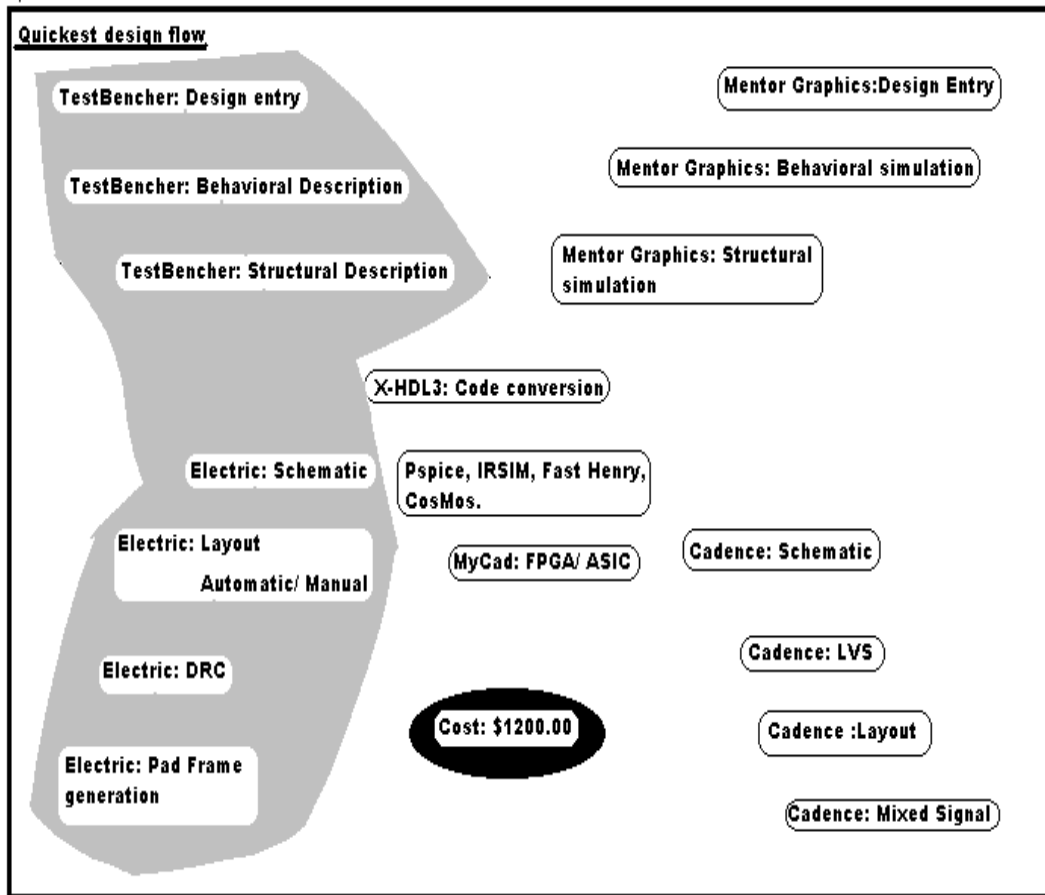


Figure 6.1: Recommended design flow for quick digital design

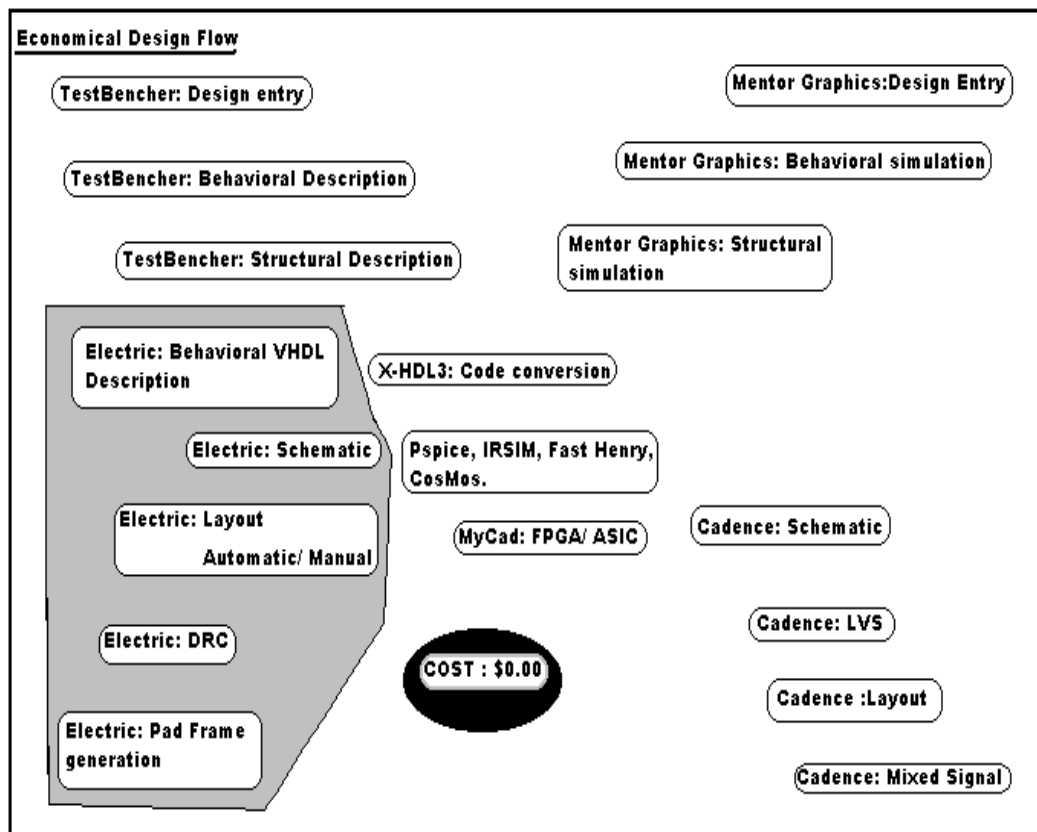


Figure 6.2: Budget oriented design flow

Table 6.2 shows a list of CAD tools and their properties that were considered for this document. The reasons for choosing TestBencher , X-HDL3 and Electric are based on the objective of this document which is listed in table 1.2.

| <u>Electric</u> | <u>X-HDL3</u> | <u>TestBencher</u> | <u>DW-2000</u> |
|-----------------|---------------|--------------------|----------------|
| Entire | | | Description |
| Design | Verilog | Front End HDI | FPGA/ASIC |
| Flow | ? VHDL | Description | Synthesis |
| \$0.00 | \$8,000.00 | \$2,500.00 | \$29,000.00 |
| FREE | Yes | Yes. | Yes |
| Compiler, | Generate | | |
| Open | Synthesizable | Automatic Test | |
| Source | code. | Bench | |
| More | Broader | | |
| interactive | VHDL Syntax | Analog w/f | |
| simulators, | Compatibility | plotting. | |

6.2 CONCLUSION

A demonstration of how inexpensive CAD tools can be used to achieve a fairly quick design flow has been shown. However, as long as commercial heavyweights exist, smaller and cheaper CAD tools find it hard to be accepted in the industry. Their credibility and compatibility is always doubted.

The design flow suggested in this document will work because of the following reasons. Firstly, Tektronics and Agilent Technologies are using TestBench. Secondly, Electric was used by groups in Sun Microsystems, and has been successfully implemented in taping out full custom ICs. These CAD tools are industry compatible and the digital design flow works for commercial development. This design flow would give universities a chance to do VLSI research not just at a theoretical level, but also at a wider commercial perspective.

Future Work:

The design flow suggested in this document needs to be verified by successfully taping out at least one custom IC.

Electric is has been an encouraging find. It is fairly advanced and is free. Since it is open sourced it provides a channel for improvement and customization.

Simulators in electric are not very interactive and lot of work can be done on improving them.

The Silicon Compiler is quite old and can route only in two metal layers. New bright ideas can be implemented by enhancing the Silicon Compiler (say, routing in 6 metals).

Currently Electric supports only VHDL. It could be improved by incorporating a Verilog compiler. Writing a Verilog compiler is difficult, however, a syntax mapping application that can emulate a Verilog deck could be written.

An RTL synthesizer for Electric needs to be developed for Electric. Currently, a synthesizer using neural network algorithms is being developed in University of New Mexico.

Keeping up with the spirit of GNU, large numbers of standard cells made by different teams in different universities can be shared, thus making a huge database of digital cells. These cells could be used to build larger digital circuits or programmable mixed signal processors.

Lastly, efforts should be made to make Electric more versatile and acceptable by the industry by suggesting to vendors to sell standard Libraries in readable dumps that can be read by any CAD tool.

APPENDIX

APPENDIX A: BEHAVIORAL DESCRIPTION OF SAR

Given below are the simulation results and the Verilog used to simulate the SAR circuit at a behavioral level. Table A1 shows the behavioral description of the SAR circuit. Table A2 shows the VHDL file generated by X-HDL3 with table A1 as input.

```
module sartheory (d,soc,rst);  
input soc,rst;  
output [7:0]d;  
reg [7:0]d;  
integer i;  
integer vin;  
initial  
begin  
i=7;  
vin=123;  
end  
always @ (posedge rst)  
begin  
d=7'b0000000;  
end
```

Table A1: Behavioral description of SAR in Verilog. (continued)

Table A1: (continued)

```
always @ (posedge soc)
begin
while (i > 0)

begin
d[i]=1'b1;
if(d > vin)

begin
d[i]=1'b0;
end
i=i-1;
end
end
endmodule
```

--VHDL code for figure 2.4

```
-----
-- Design name      : sartheory
-- Author           : sagar
-----
```

```
--
ENTITY sartheory IS
PORT (
d          : OUT std_logic_vector(7 DOWNT0 0);
soc        : IN std_logic;
rst        : IN std_logic);
```

Table A2: VHDL file generated by X-HDL3 (continued)

Table A2: (continued)

```

END sartheory;
ARCHITECTURE translated OF sartheory IS
    SIGNAL i          : integer;
    SIGNAL vin        : integer;
    SIGNAL d_xhdl1    : std_logic_vector(7 DOWNTO 0);
BEGIN
    d <= d_xhdl1;

    PROCESS
        VARIABLE xhdl_initial : BOOLEAN := TRUE;
    BEGIN
        IF (xhdl_initial) THEN
            i <= 7;
            vin <= 241;
            xhdl_initial := FALSE;
        ELSE
            WAIT;
        END IF;
    END PROCESS;
    PROCESS
    BEGIN
        WAIT UNTIL (rst'EVENT AND rst = '1');
        d_xhdl1 <= "00000000";
    END PROCESS;
    PROCESS
    BEGIN
        WAIT UNTIL (soc'EVENT AND soc = '1');
        WHILE (i > 0) LOOP
            d_xhdl1(i) <= '1';
            IF (d_xhdl1 > to_stdlogicvector(vin, 8)) THEN
                d_xhdl1(i) <= '0';
            END IF;
            i <= i - 1;
        END LOOP;
    END PROCESS;
END translated;

```

Table A2: VHDL file generated by X-HDL3

In figures A1 to A4, the simulation results for $V_{in} = '44'$ for the SAR circuit (Verilog code in table A1) are shown.

In the following figures,

V_{in} is a scaled version of 5 volts. Where $0v=0$; $2.5v=128$ and $5v=255$

Register $d[7:0]$ holds the result.

Signals 'rst', 'soc' represent 'RESET' and 'Start Of Conversion' respectively.

'Eoc' represents, 'End Of Conversion'. When 'Eoc' is asserted, the bus $d[7:0]$ holds a valid the result that can be sampled. But for an abstract simulation, details like Eoc are neglected.

The value of the register when Soc is asserted shows that $d[7:0] = \text{hex } 44$ and $\text{vin} = \text{hex } 44$.

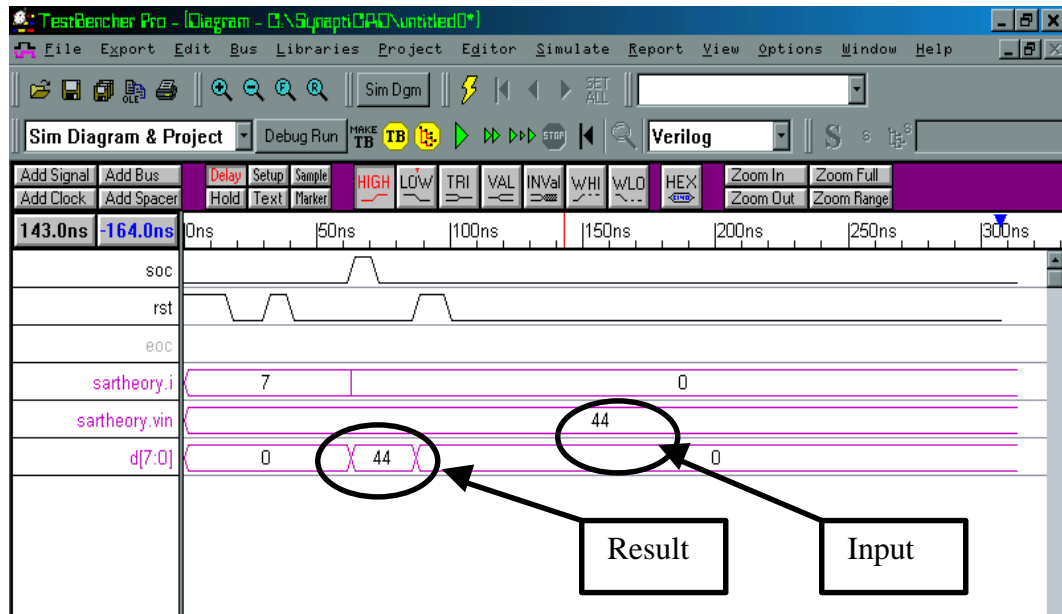


Figure A1: $d[7:0]$ shows the result for input voltage of 44.

Given Below is the simulation result with the bus d[7:0] expanded

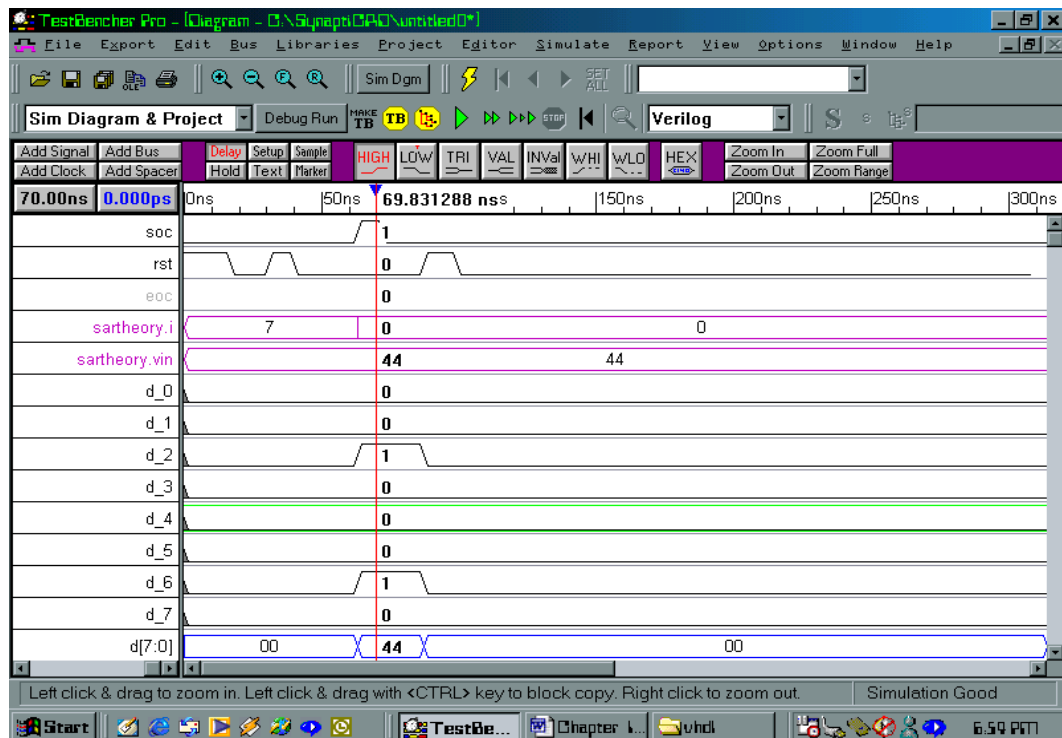


Figure A2: Result with bus d[7:0] expanded

The SAR has a probability of 1 bit error 50% of the times. The following simulation illustrates this property.

Vin = Hex F1;

d[7:0]= Hex F0;

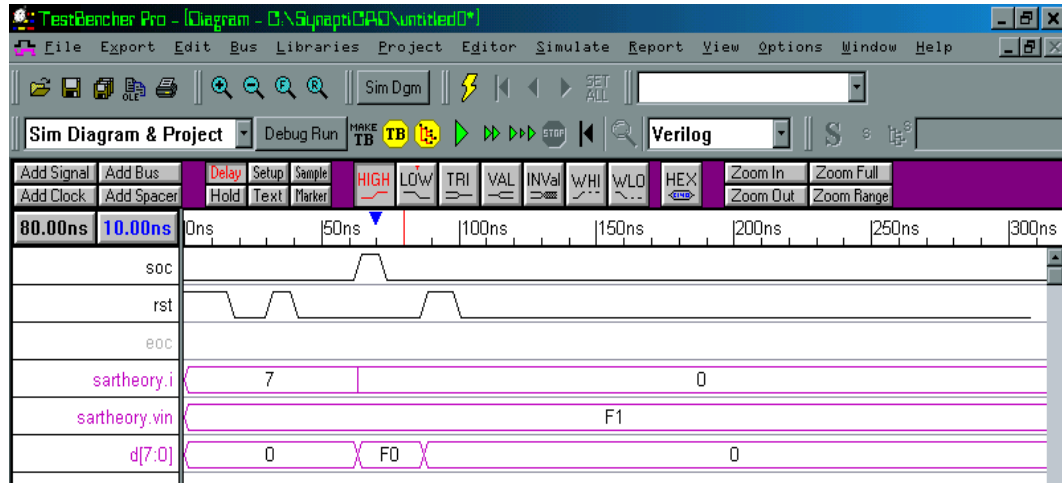


Figure A3: Result showing the expected 1bit error

Result with the bus d[7:0] expanded.

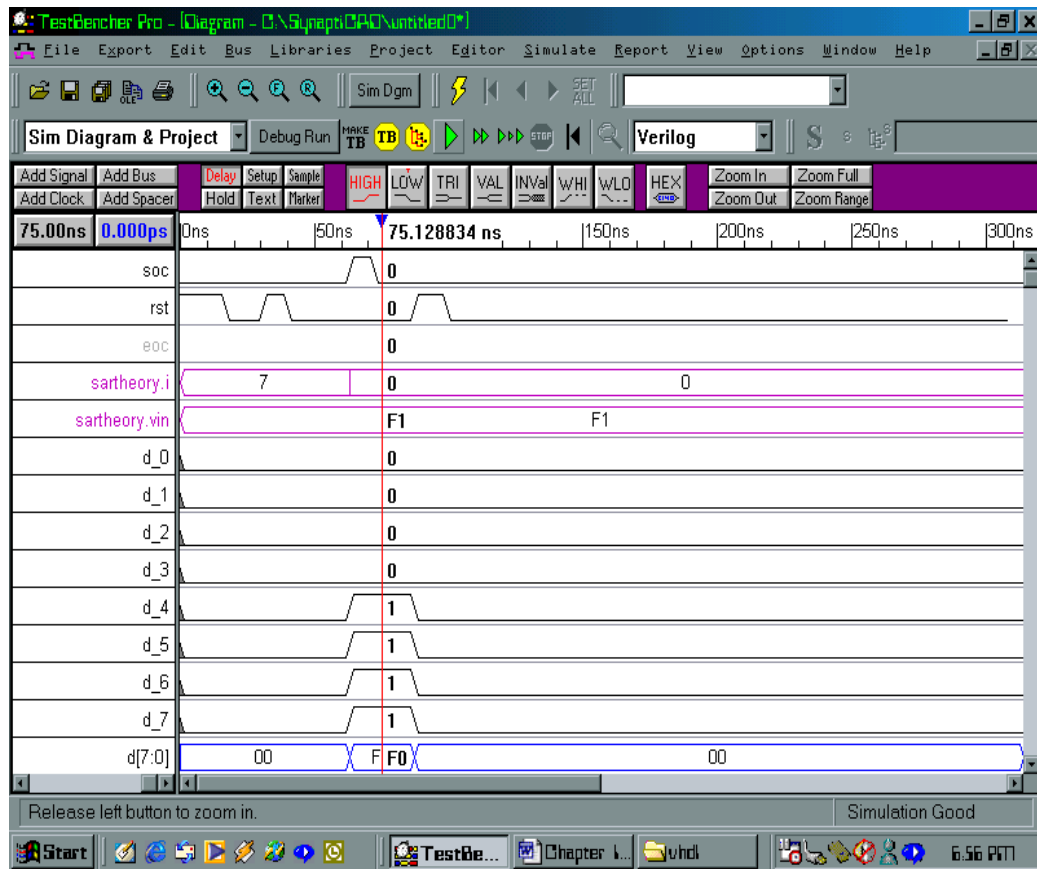


Figure A4: Result with bus d[7:0] expanded

APPENDIX B: STRUCTURAL DESCRIPTION OF SAR

Given below are the simulation results and Verilog files associated with each module of the SAR circuit. Figure B1 shows the Verilog description of a dfflipflop.

Verilog code for dff

```
module dff (q,qbar,d,rst,clk);
input d,clk,rst;
output q,qbar;
wire s,r,t,u,rstbar,din;
not (rstbar,rst);
and g5 (din,d,rstbar);
nand g4 (t,r,din);
nand g3 (r,t,clk,s);
nand g2 (s,u,clk);
nand g1 (u,s,t);
nand g7 (q,qbar,s);
nand g8 (qbar,q,r);
endmodule
```

Table B1: Verilog description of dff

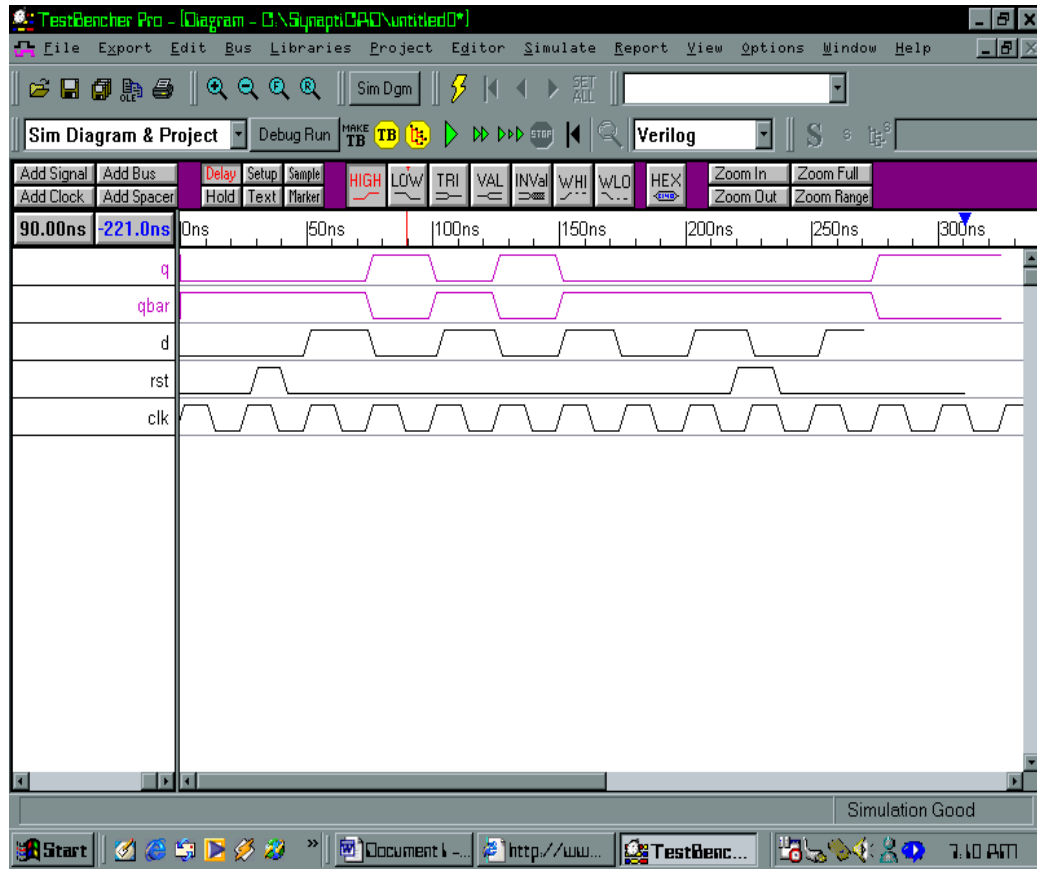


Figure B1: Simulation result of dff

Clock divider circuit:

```
module clkdiv (d,rst,soc,clk,eoc,socout);
input rst,clk,soc;
output [31:0]d;
output eoc,socout;
wire [32:0]dbar;
wire a,b,c,e,f;
dff (a,b,soc,rst,clk);
dff (socout,e,a,rst,clk);
dff (c,f,socout,rst,clk);
dff (d[0],dbar[0],c,rst,clk);
dff (d[1],dbar[1],d[0],rst,clk);
dff (d[2],dbar[2],d[1],rst,clk);
dff (d[3],dbar[3],d[2],rst,clk);
dff (d[4],dbar[4],d[3],rst,clk);
dff (d[5],dbar[5],d[4],rst,clk);
dff (d[6],dbar[6],d[5],rst,clk);
dff (d[7],dbar[7],d[6],rst,clk);
dff (d[8],dbar[8],d[7],rst,clk);
dff (d[9],dbar[9],d[8],rst,clk);
dff (d[10],dbar[10],d[9],rst,clk);
dff (d[11],dbar[11],d[10],rst,clk);
dff (d[12],dbar[12],d[11],rst,clk);
dff (d[13],dbar[13],d[12],rst,clk);
dff (d[14],dbar[14],d[13],rst,clk);
dff (d[15],dbar[15],d[14],rst,clk);
dff (d[16],dbar[16],d[15],rst,clk);
dff (d[17],dbar[17],d[16],rst,clk);
dff (d[18],dbar[18],d[17],rst,clk);
dff (d[19],dbar[19],d[18],rst,clk);
dff (d[20],dbar[20],d[19],rst,clk);
dff (d[21],dbar[21],d[20],rst,clk);
dff (d[22],dbar[22],d[21],rst,clk);
dff (d[23],dbar[23],d[22],rst,clk);
dff (d[24],dbar[24],d[23],rst,clk);
dff (d[25],dbar[25],d[24],rst,clk);
dff (d[26],dbar[26],d[25],rst,clk);
dff (d[27],dbar[27],d[26],rst,clk);
dff (d[28],dbar[28],d[27],rst,clk);
```

Table B2: Verilog description of clock divider (continued)

Table B2: (continued)

```
dff (d[29],dbar[29],d[28],rst,clk);  
dff (d[30],dbar[30],d[29],rst,clk);  
dff (d[31],dbar[31],d[30],rst,clk);  
dff (eoc,dbar[32],d[31],rst,clk);  
endmodule
```

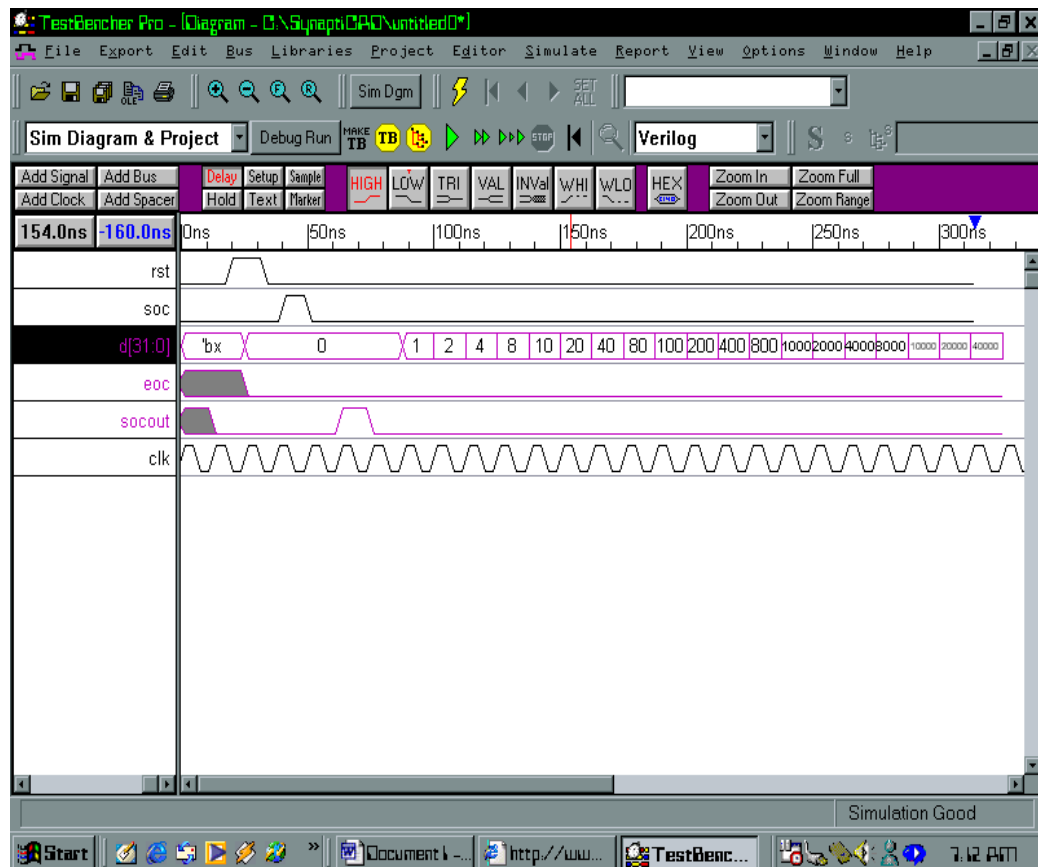


Figure B2: Simulation result of clock divider

Result with d[31:0] expanded: Note that Socout becomes Eoc.

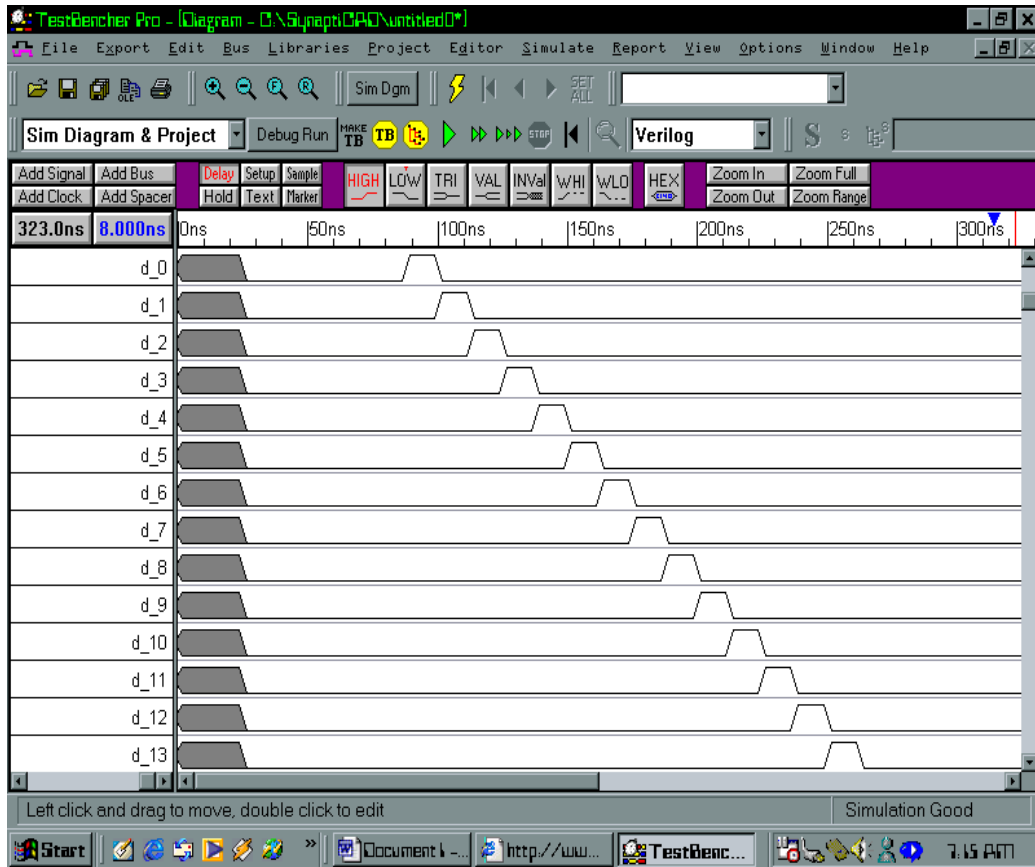


Figure B3: Simulation result of clock divider with bus expanded

Verilog code for Logic1 block:

```
module logic1 (clkout,rst,soc,clk,eoc);
input clk;
input rst,soc;
output eoc;
output [7:0]clkout;
wire [7:0]clkout1;
wire [31:0]w;
wire socout;
clkdiv g1 (w,rst,soc,clk,eoc,socout);
or (clkout1[7],w[0],w[2]);
or (clkout1[6],w[4],w[6]);
or (clkout1[5],w[8],w[10]);
or (clkout1[4],w[12],w[14]);
or (clkout1[3],w[16],w[18]);
or (clkout1[2],w[20],w[22]);
or (clkout1[1],w[24],w[26]);
or (clkout1[0],w[28],w[30]);
or (clkout[0],clkout1[0],socout);
or (clkout[1],clkout1[1],socout);
or (clkout[2],clkout1[2],socout);
or (clkout[3],clkout1[3],socout);
or (clkout[4],clkout1[4],socout);
or (clkout[5],clkout1[5],socout);
or (clkout[6],clkout1[6],socout);
or (clkout[7],clkout1[7],socout);
endmodule
```

Table B3: Verilog description of logic block 1

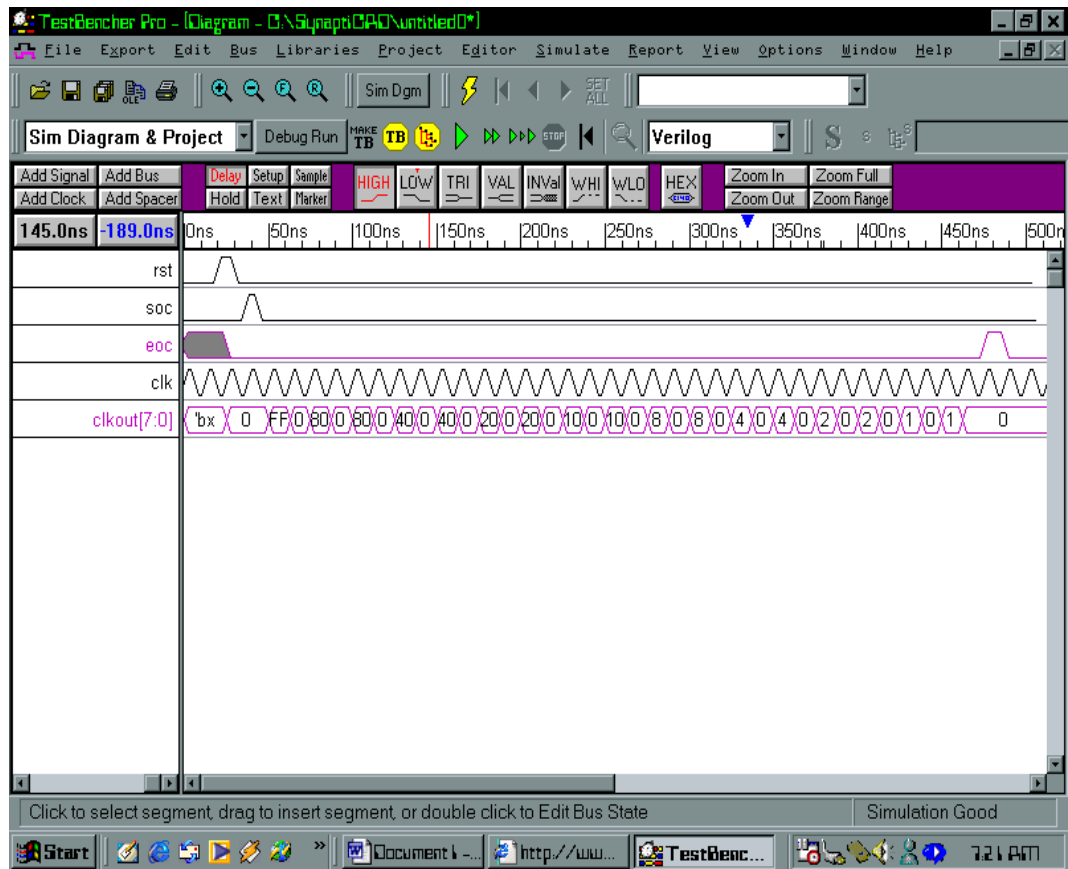


Figure B4: Simulation result of logic block 1

Notice 'eoc' goes high after the conversion .

'Clkout' feeds the main register with deferred clock edges.

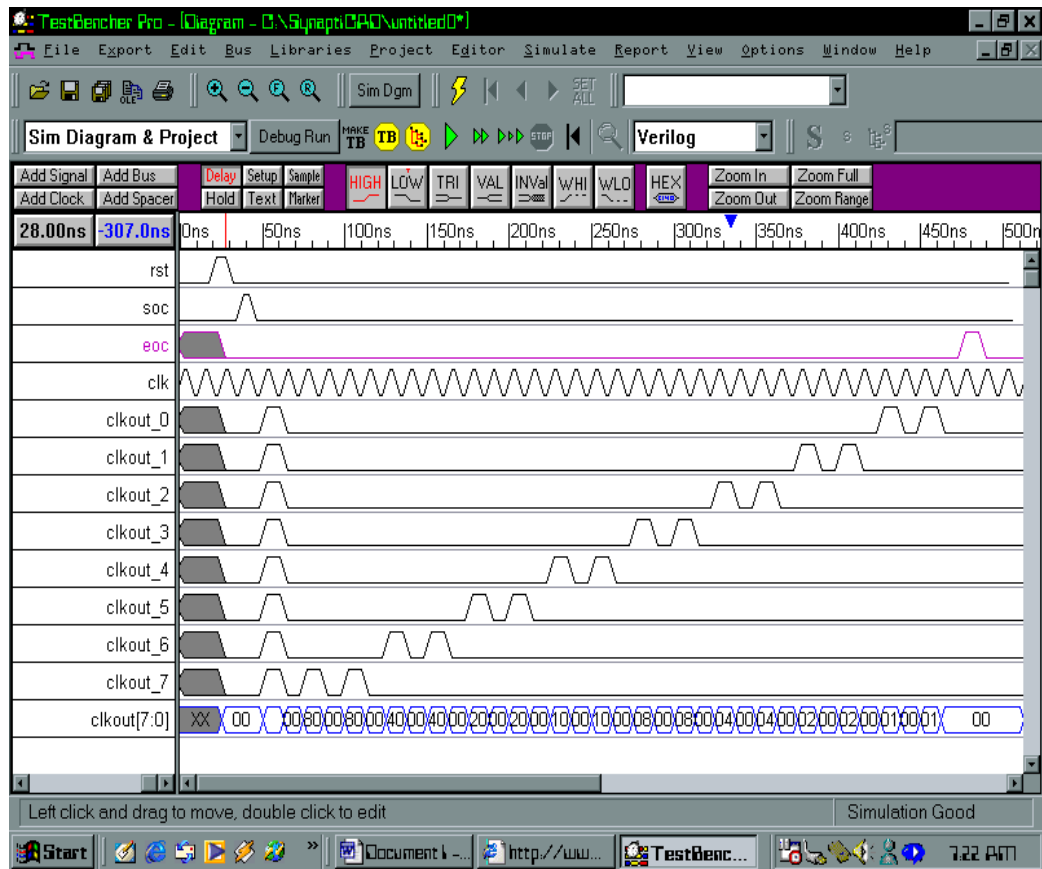


Figure B5: Simulation result with bus d[31:0] expanded

Main register:

Verilog code

```
module mainreg (clkin,din,dout,rst,soc,clk);
input [7:0]clkin;
input rst,soc,clk;
input [7:0]din;
output [7:0]dout;
wire [7:0]doutbar;
wire a,b,c,e;
wire rst1;
or (a,rst,soc);
dff (rst1,b,a,rst,clk);
dff (dout[0],doutbar[0],din[0],rst1,clkin[0]);
dff (dout[1],doutbar[1],din[1],rst1,clkin[1]);
dff (dout[2],doutbar[2],din[2],rst1,clkin[2]);
dff (dout[3],doutbar[3],din[3],rst1,clkin[3]);
dff (dout[4],doutbar[4],din[4],rst1,clkin[4]);
dff (dout[5],doutbar[5],din[5],rst1,clkin[5]);
dff (dout[6],doutbar[6],din[6],rst1,clkin[6]);
dff (dout[7],doutbar[7],din[7],rst1,clkin[7]);
endmodule
```

Table B4: Verilog description of main register

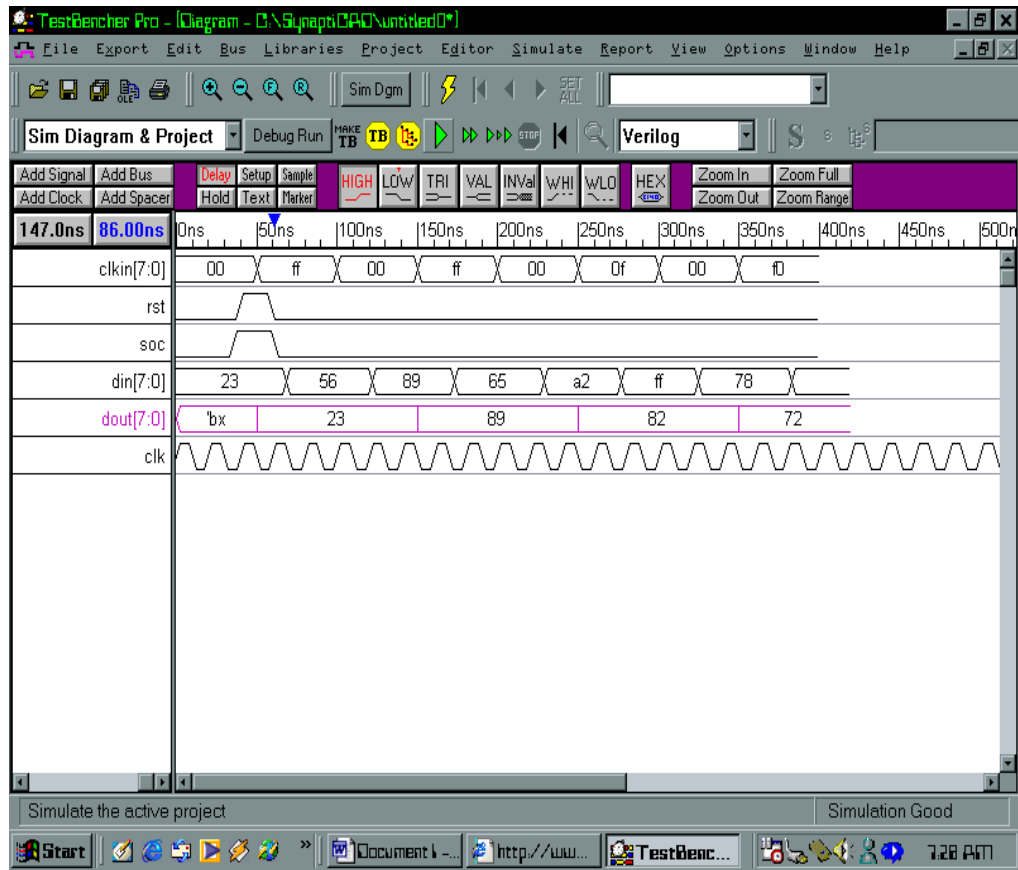


Figure B6: Simulation result of main register

Complete Digital part of SAR A/D.

Verilog code

```
module digsar (d,eoc,rst,soc,gt,clk);  
input rst,soc,gt,clk;  
output eoc;  
output [7:0]d;  
wire [7:0]w1;  
wire [7:0]w2;  
logic1 (w1,rst,soc,clk,eoc);  
logic4 (d,rst,gt,w1,soc,clk);  
endmodule
```

Table B5: Verilog description of digital part of SAR.

simulation showing the main register value converge to 'ff'. Note that 'gt' is the signal that is assumed to be coming from the comparator.

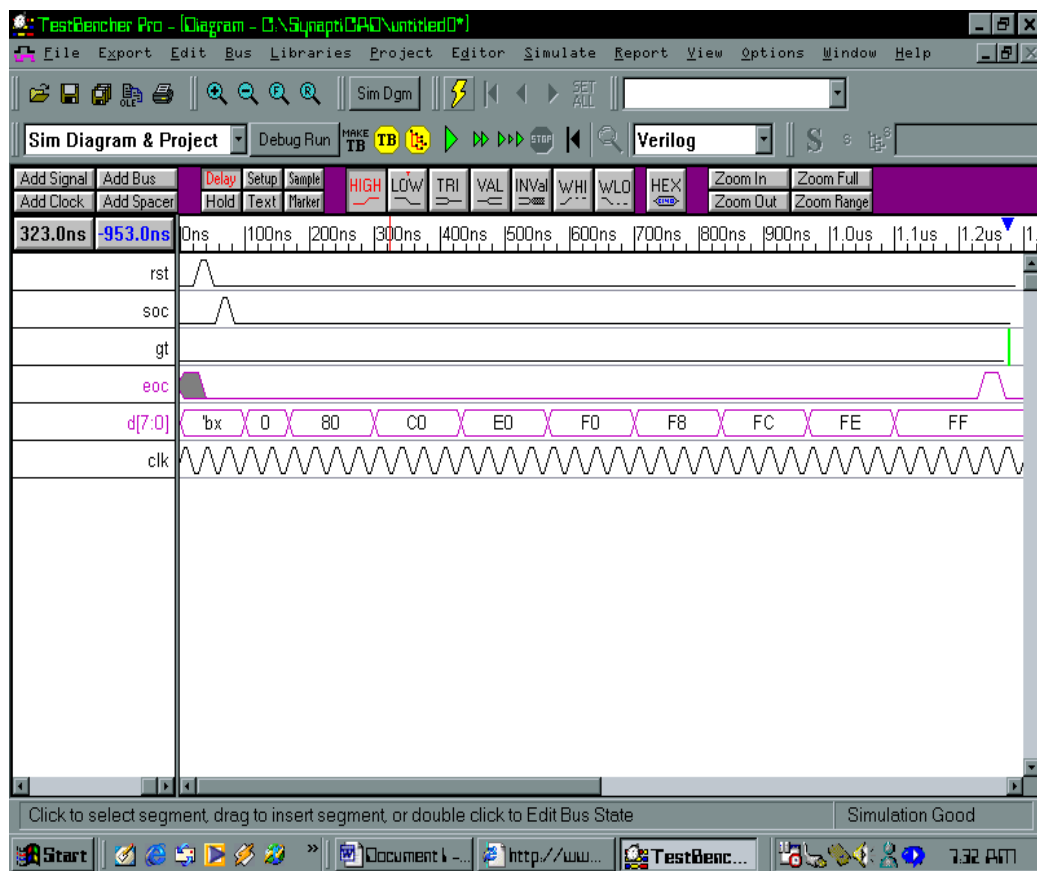


Figure B7: Simulation of digital part of SAR.

Simulation showing the main register value converge to '4f'.

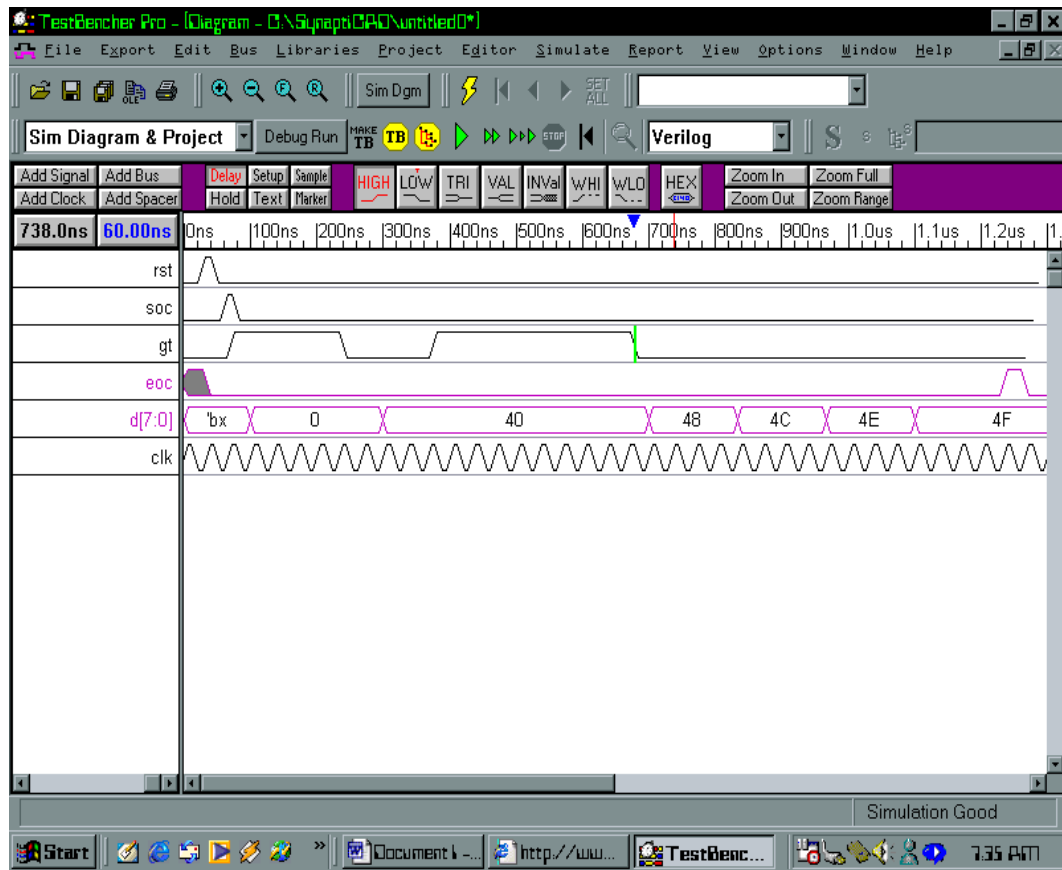


Figure B8: Simulation of digital part of SAR.

Mychip:

This is a description of the complete A/D with the analog part replaced by a behavioral description.

```
module mychip (d,rst,soc,eoc,clk,gt);  
input rst,soc,clk;  
output [7:0]d;  
output eoc;  
output gt;  
d_to_a (gt,d,clk);  
digsar (d,eoc,rst,soc,gt,clk);  
endmodule
```

Table B7: Verilog description of dff

The input value of the analog voltage was specified to be '47'.
 The output obtained was '48'. Note that SAR type D/A always have a 50 % chance of 1 lsb error.

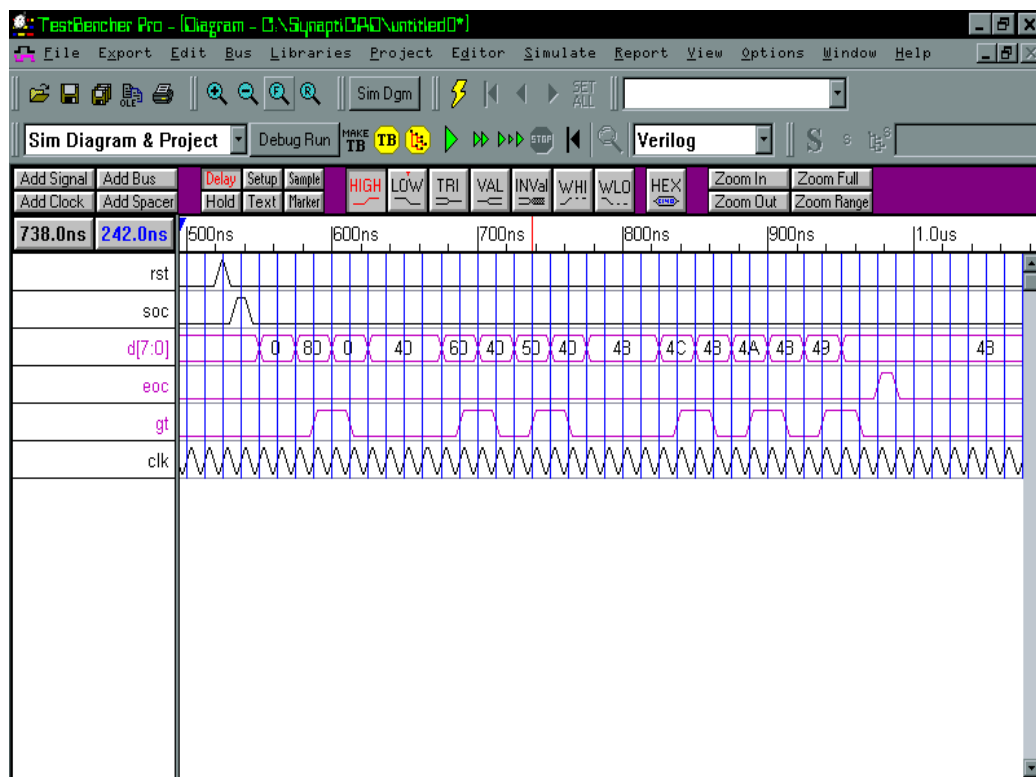


Figure B9: Simulation result of MYCHIP

Simulation for $V_{in} = f_6$

$V_{out} = f_6$

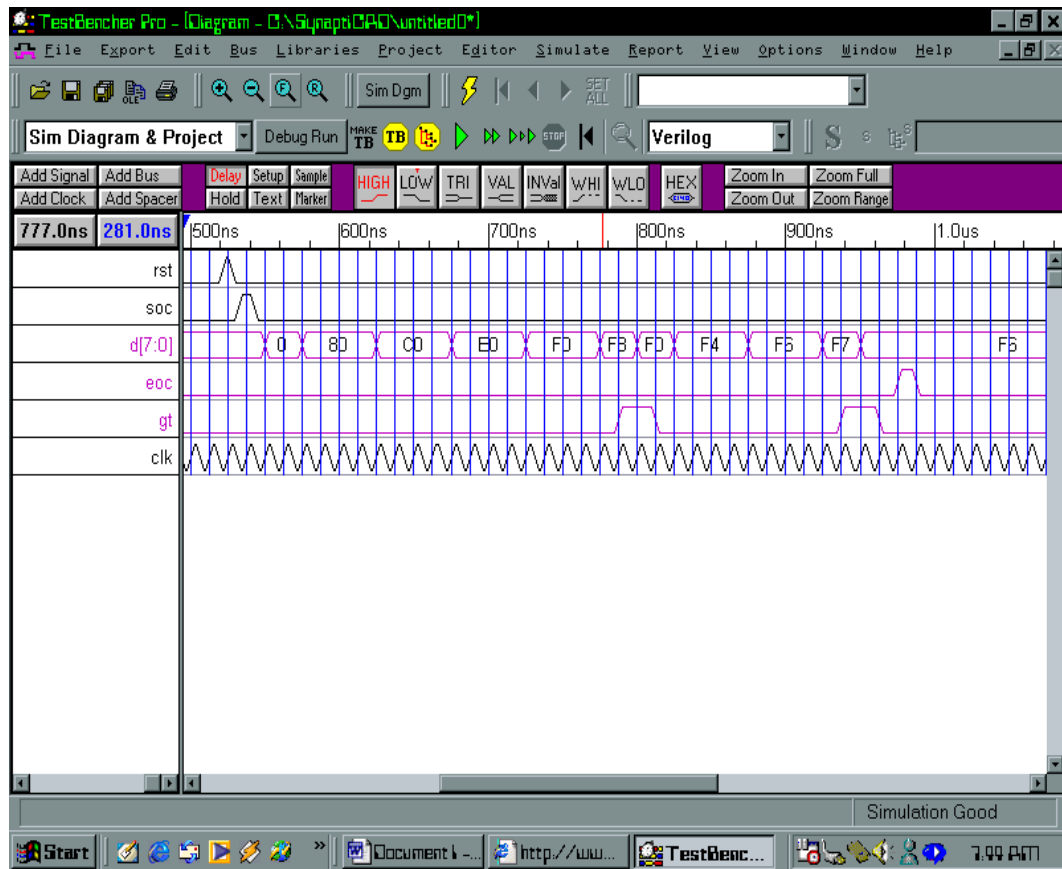


Figure B10: Simulation result 2 of MYCHIP

The output is sampled whenever eoc is asserted

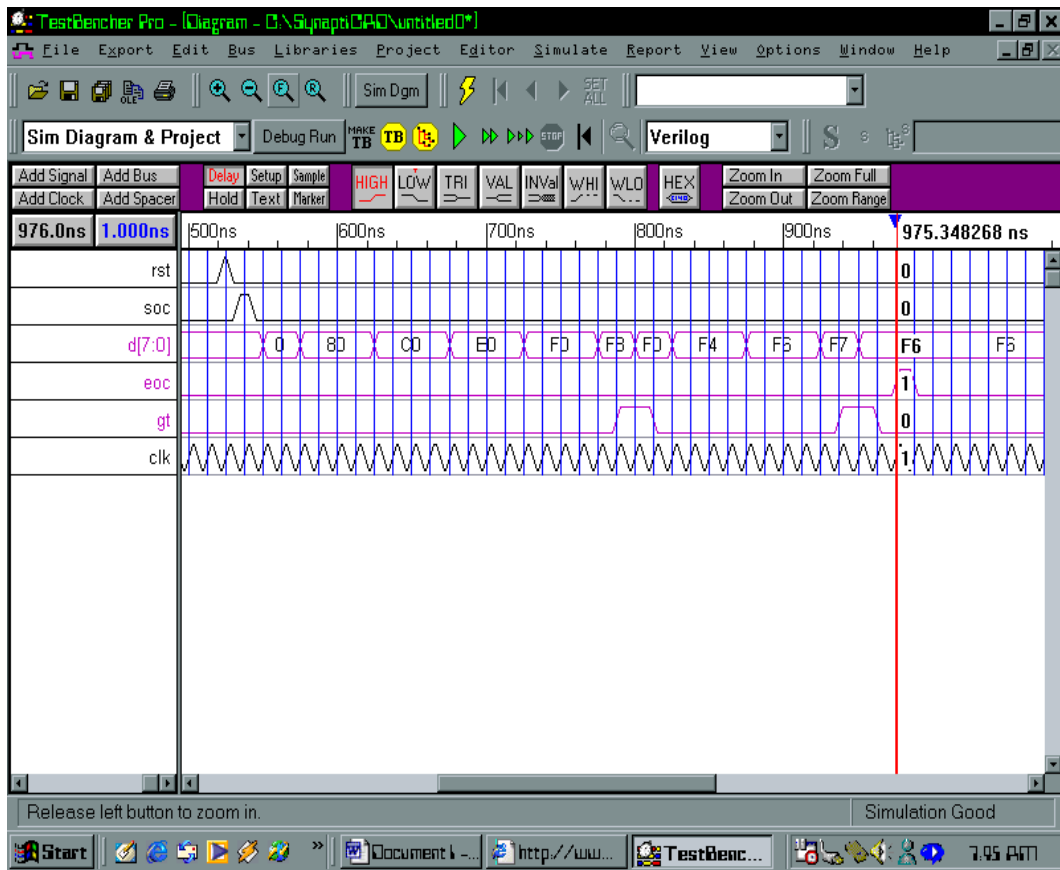


Figure B11: Simulation result 3 of MYCHIP