Anil Lakshman Bahuman
An Evolutionary Approach to Standard Cell Design Automation
(Under the direction of Benjamin Bishop)

The problem of designing the transistor-level layout of cells in a standard cell library is a multi-objective design optimization problem. Contemporary methods are optimization or compaction engines that rely on a schematic representation supplied by a design engineer. We have demonstrated the possibility of applying a modified genetic algorithm (GADO) to design a cell given only a behavioral description. A working inverter is designed as a proof of concept along with other inverters with arbitrary label placements. This is an example of design without human intervention, i.e. a *computer* design as opposed to a *computer-aided* design. The complete set of experimental results may be found at the project web site `http://james.cs.uga.edu`.

Index words: VLSI, design automation, standard cell, genetic algorithm, GADO, MAGIC, EDA

An Evolutionary Approach To Standard Cell Design Automation

by

Anil Lakshman Bahuman

B.E., Karnataka Regional Engineering College, Surathkal, India, 1998

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2001

An Evolutionary Approach To Standard Cell Design Automation

by

Anil Lakshman Bahuman

Approved:

Major Professors: Benjamin Bishop
Khaled Rasheed

Committee: Ron McClendon
Richard Marsh

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
December 2001

Dedication

To my grandfather, Krishnoji Rao for his confidence in me,

To my father, Commander Lakshman for his vision,

To my mother, Revathi for her support,

To my sister, Komal for her love and affection.

Acknowledgments

Success is rarely an individual achievement. The idea of using genetic algorithms for VLSI (Very Large Scale Integration) standard cell design came from discussions between Drs. Benjamin Bishop and Khaled Rasheed. They planned to sponsor a research assistantship in the area and that is how I entered the picture.

Thus, foremost, I would like to thank Benjamin and Khaled for making this possible. Their guidance throughout the project has been invaluable. I would also like to thank Don Potter, Ron McClendon, Michael Covington, Takoi Hamrita, Richard Marsh, Angela Paul, Jean Power, Nelson Rushton, Fred Maier, Vassilka Deltcheva, Swaroop Vattam, Sanjay Chellapilla, Abhisek Jain, Dev Ashish and Cartic Ramakrishnan for their support of this work.

This work was supported by Georgia State's Yamacraw initiative under the following grants: GCOMM2001.690130501-GIT and GCOMM2002.6900000106-GIT.

Preface

Computers have been assisting humans in design of artifacts for several years now. But with the complexity of engineering design acquiring levels that baffle individual human minds, there is a cry for heuristic automation methods that can suggest designs allowing the human to add only final touches. An example of the need for such techniques is in the area of the design of integrated circuits. VLSI technologies have reached a complexity that is mind-boggling. Electronic design automation (EDA) tools allow the human designer (electrical engineer) to work with high level descriptions. We demonstrate the possibility of one such tool that can assist engineers to build standard cells on-the-fly customized to criteria specified by them. Our method is an example of design by evolution by the use of genetic algorithms.

The field of electronic design automation poses one of the toughest challenges to the evolutionary computation community.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This section has been written for a reader new to Very Large Scale Integration (VLSI). If you are already familiar with standard cells, MAGIC and SPICE you may jump to page 7.

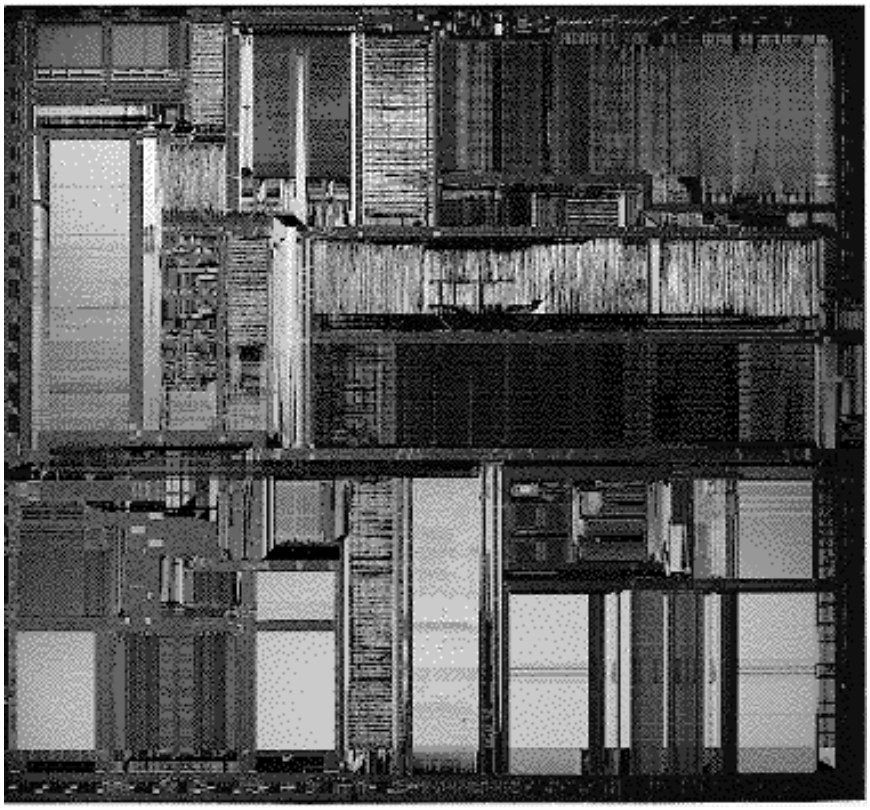

Figure 1.1: Inside the Intel Pentium

1.1 VLSI Design

The Electronic Design Automation (EDA) industry provides the critical technology to design electronics that support the Information Age, including: communications, computers, space technology, medical and industrial equipment and consumer electronics. Computers have aided design engineers for a long time. They have been used by designers to create, modify, test and store their designs. Various tools have been developed to assist the designer but the design process has essentially been human-driven. Computer chips, for example, used to be designed laboriously by hand, component by component. With the ever increasing complexity of computer chip design, it was natural for automation to play an important role. The easier tasks such as generating mask specifications for fabrication were first automated by deterministic algorithms. The increasing complexity of Very Large Scale Integrated Circuits (VLSI) — the technology of putting more than 100,000 transistors on a single chip — makes the design search space intractable and requires heuristic techniques that negotiate the combinatorial challenges to design high quality chips in an ever shortening time-to-market. Various heuristic methods [11] have been employed in chip design with a good number of successes.

The design of VLSI chips is a multi-objective design optimization problem. One unpopular approach is *flat* device-level layout synthesis where the entire design is created in a single step by working in a component by component fashion. According to Burns and Feldman [6]:

> This approach handles design tuning naturally [i.e. the ability to optimize a circuit by changing device parameters - Anil]. However, the synthesis phase is difficult algorithmically. Both the device-level placement and device-level routing problems are NP- complete [11]. Near-optimal

algorithms are feasible for small designs only and fast heuristics handling large designs yield poor quality layouts.

The inherent complexity is addressed by another approach — a hierarchical design process where a higher process can work by placing components treated as black-boxes. These black-boxes are designed by another design process that works with components at a lower level of abstraction. Thus there are various design processes each working at a different level of abstraction. Some of the names associated with these processes are circuit partitioning, cell placement, cell routing, compaction etc. Each process depends on the other and a process at one level may not have complete information about components at a lower level and hence cannot use it in its cost function and has to make approximations.

This field poses one of the toughest challenges to the evolutionary computation community (and others interested in design optimization) since large-scale, coupled optimization problems commonly arise. Genetic algorithms have been used for various problems in VLSI. Mazumder and Rudnick [12] is a good survey of such applications. We have focused our attention on an area where we are not aware of any GA based solutions — that of standard cell design.

At the lowest level of logic, many designers work with standard cells. The idea here is to develop a library of frequently used logics to speed up the design process so that a new chip does not have to be built all the way down to the transistor level. Higher level tools such as standard cell, datapath placement and routing tools can then request cells from the library and complete the chip design in days instead of weeks. But then someone has to create and maintain a library of standard cells.

1.2 Standard Cell Design

The design of a standard cell is challenging. At this level, designers work with transistors that are in turn made from bringing certain materials in contact with one another. Figure 1.2 shows how a transistor is made by bringing a polysilicon

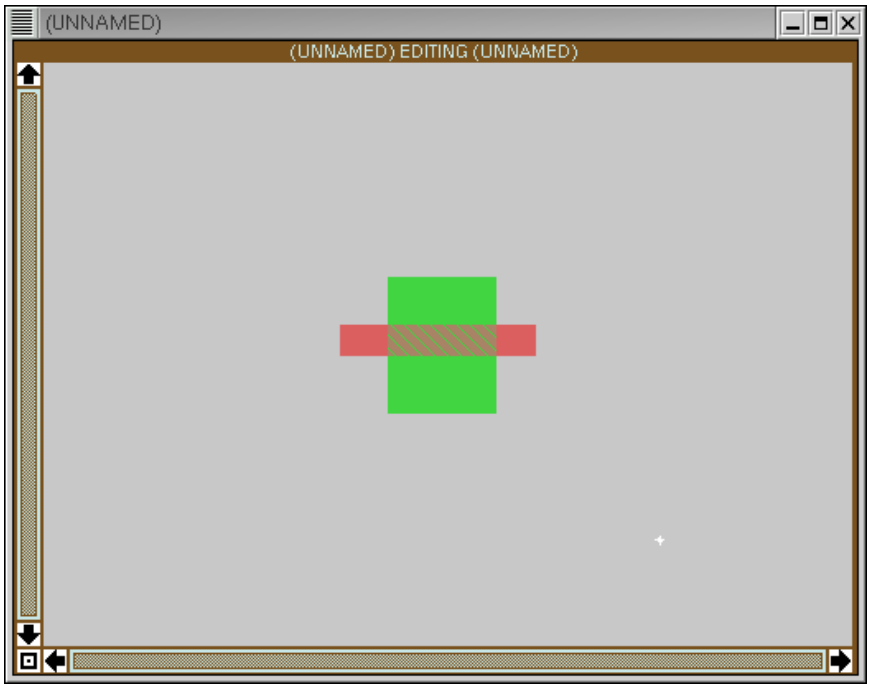

Figure 1.2: A transistor designed in MAGIC

layer in contact with a diffusion layer. What you see is a snapshot from MAGIC [18], a layout editor which is essentially a smart tool that works with colored polygons that represent physical layers that form the complex circuitry on a chip. It is smart as it knows that bringing a polysilicon layer (the narrow rectangle) in contact with an n-diffusion layer (the wider rectangle) creates an n-transistor which is shown as a hatched region in the figure. The tool is aware of the electrical characteristics of the materials and has been programmed with a set of design rules for a given process. We say *for a given process* because these rules can change as the technology used by

the foundry (which eventually etches the electronic copy of the design onto silicon) changes. Design rules are constraints on the placement of the polygons that reflect the practical constraints (such as tolerance levels) that the foundry (manufacturers of the chip) has to deal with.

An example of a design rule would be — *A red polygon (polysilicon) has to be at least 3 lambda away from another red polygon (polysilicon)* where *lambda* is a unit of measurement. This unit actually translates to a physical measure (usually in nanometers) depending on the process. A software tool called the design rule checker (DRC) may be called into action within MAGIC to check if all the polygons placed so far meet all design rule constraints. Areas that do violate these constraints are filled with white boxes (Figure 1.3) so as to alert the designer that the polygons need to be rearranged since this design is not practically feasible. Other design rules constrain the shapes and minimum sizes of certain polygons.

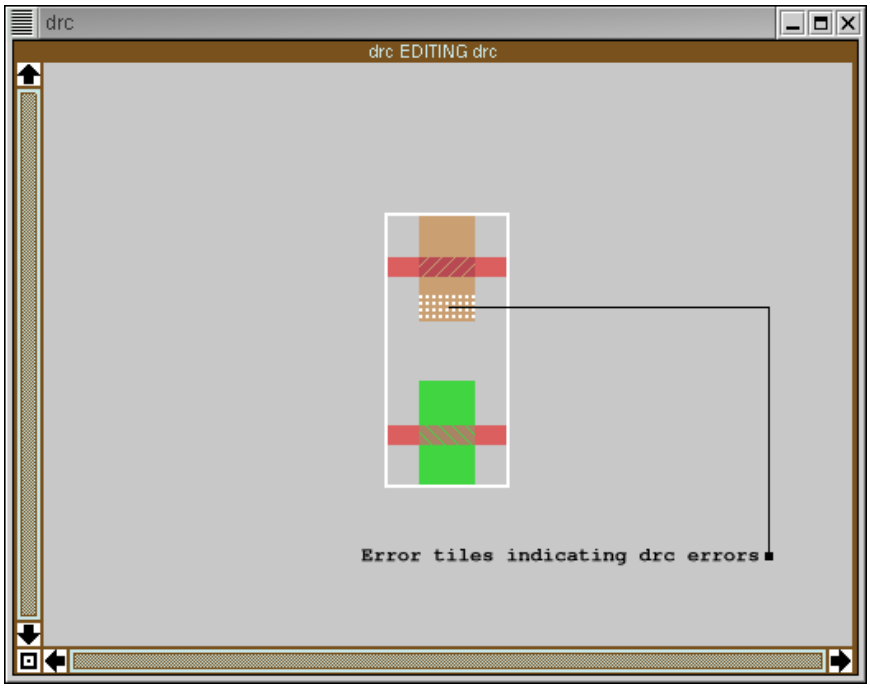

Figure 1.3: DRC errors highlighted by MAGIC

Transistors may be connected by blue polygons that represent conducting metals and hence wires. Once the designer has finished placing and connecting the transistors for a circuit she had in mind (or represented as a circuit schematic on paper or electronic form), she does a final check to make sure there are no design rule errors. She now goes on to check if the circuit does behave as intended. In the example in Figure 4.9 of an inverter (also called NOT gate), the expected behavior is to invert the given input, i.e. a "zero" at the input will produce a "one" at the output and vice versa.

This simulation is conducted by another tool called a circuit simulator. We used SPICE [19, 20] to test our designs. SPICE can take a file extracted from MAGIC , supply the inputs specified by the user and generate the outputs requested by the user. Figure 1.4 shows the waveforms generated (as viewed by another tool called `nutmeg`). The tool displays these in color but since the reader may be viewing the snapshot in black and white, additional labels *in* and *out* have been placed all along the waveforms for your reference. The input waveform goes from 3.3 V, down to 0 V, up to 3.3 V and down again to 0 V. The output waveform does the exact opposite. It goes from 0 V to 3.3 V to 0V and back to 3.3 V. Note that there are time delays experienced by the output waveform and it does seem erratic at places but what is important is that the values be correct at certain time instants or samples. In this case we may look at the values at 10 ns and 30 ns. Note that at these instants the circuit behaves obediently: if the input is at 3.3 V (binary *one*) the output is at 0 V (binary *zero*) and vice-versa. The values at both samples in Figure 1.4 are thus *inverted* which is the intended behavior of the circuit.

Once the designer is convinced that the circuit behaves correctly she goes on to tune certain performance criteria that are of interest such as (1) area of the design, (2) the delay in responding to change in inputs and (3) the power dissipated by the devices and wires. The first is measured in MAGIC and the last two are measured

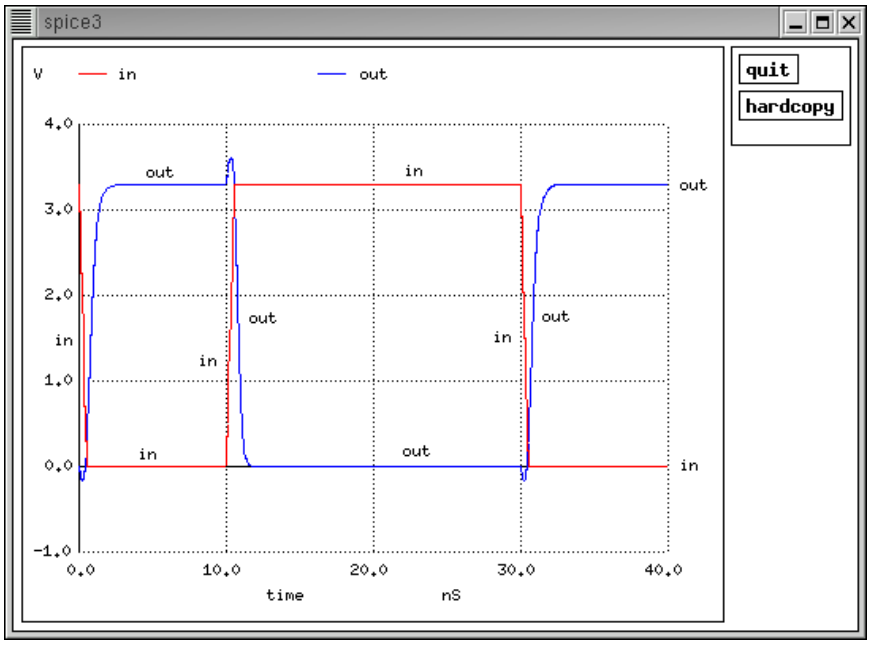

Figure 1.4: Input and Output Waveforms for the inverter in Figure 4.9

in SPICE. Changes are made in the design and the entire process is repeated until the circuit is optimized for the task in hand. Note that optimizing for the criteria (1, 2 and 3 above) involve design tradeoffs. Also note that an attempt to automate all these processes without human intervention will also have to find ways of interpreting the cues offered by MAGIC and `nutmeg` without resorting to the visual medium. This requires developing scripts that can parse the output from these tools to seek the information we need.

Traditionally a large number of these cells are stored in a library known as the standard cell library. These building blocks number in the thousands catering to the demand for frequently used logics. Examples include the inverter, NAND gates, full adder, latch etc. Adding new cells to handle constraints that were not considered at the time of cell design is difficult if not impossible. Every time the process changes, the design rules change. These libraries have to be redesigned, manually

in many cases. *Compaction* tools have been built that can work with an old library and migrate them to the new process by shifting the components to accommodate design rule changes but these have severe limitations. Compaction is the process of squeezing out all possible extra spaces between the devices in the circuit so as to minimize its area.

The high costs (both time and money) of designing and maintaining standard cells is a growing concern in the VLSI community. Many authors have proposed solutions but the automated techniques are yet to catch up with the quality of manually laid out cells [5].

1.3 Genetic Algorithms

Genetic Algorithms (GAs) [13] are a class of powerful heuristic algorithms that can search for a solution by a process similar to natural selection, i.e. the desired characteristics of the solution are enhanced by continued breeding. GAs attempt to find good solutions by randomly creating a pool of potential solutions to the problem and manipulating those solutions by using *genetic operators*. Genetic operators manipulate existing solutions to generate new solutions. Each solution is assigned a fitness value which is a numerical assessment of how well it solves the problem.

A solution encoded as a string of numbers corresponding to different features of the solution is known as an *individual*. Individuals with higher fitnesses are picked more frequently to produce new solutions using recombination operators. These operators work on multiple individuals (parents) and create new individuals (offsprings) that are a hybrid of the features of their parents. Another set of operators — mutation operators— randomly change some of these features.

Individuals with lower fitness values are replaced by individuals of higher fitness. As a result, the average fitness of the population tends to improve over time. The

fitness of the best individual is also expected to improve over time and the best individual may be chosen as a solution after several iterations.

Two variations in replacement strategies are:

1. replacement of the entire pool of individuals resulting in generational GAs
2. replacement of a small fraction of the pool of individuals (usually the weakest) resulting in steady-state GAs

Both GAs have been implemented by the author with success and the final choice depends on the application. The main advantages of GAs are:

1. Robust compared to path following methods as they maintain a much more thorough representation of the search space through a whole set of points that collaborate to find better regions.
2. Intrinsically parallel and hence conduct a number of searches in parallel.
3. Easy to parallelize on networks of workstations.

Chapter 2

Literature survey and Research Goal

As noted earlier there is a progression toward more automated cell synthesis techniques. Some authors [5, 6] advocate a shift in the use of standard cell libraries as we know them. The idea here is to engineer standard cells on demand or "on-the-fly" thus eliminating the need for developing and maintaining expensive static cell libraries. This would allow the designer to demand cells that may not exist in any pre-defined library, optimized for a given criterion depending on the context. In particular this would permit [10]:

1. Standard cell and datapath placement and routing tools to request cells with exact pin-orderings.

2. Logic synthesis tools to request specific logic decompositions.

3. Interconnect optimization tools to request cells with specific input and output impedance values.

4. Power optimization tools to request specific power/delay trade-offs, perhaps even specify a logic family.

Other advantages include:

1. Synthesis of cells of desired shape and size.

2. Transistor level tuning.

3. Ability to jointly conduct circuit and process optimization across process and layout architecture spaces [9].

Typical techniques are schematic dependent. They either start from a netlist of transistors (a representation showing how the transistors are connected to one another) with the transistor type and sizing specified by the designer or from some other type of a symbolic layout followed by compaction. Existing compaction techniques are unable to change the shape or orientation of layout objects and thus depend on the designer's specification. Yet another technique used is the development of procedural module generators that are programs that hard code procedures for assembling a cell as well as design rule constraints. Hard coding limits their use in the face of changes in cell architecture and interconnect technology [5].

Fixed libraries make device level tuning impossible (or difficult). Poor timing, area or power tradeoffs may result since the design of the cell is decoupled from the constraints imposed by higher-level tools. Schematic independence and device-size tuning is best accomplished via on-the-fly leaf cell synthesis. The use of C5M to develop a 440 MHz processor for IBM discussed in [6] is a good example of such a design methodology. Lefebvre, Marple and Sechen [5] note that layout synthesis tools need to optimize across all of the following phases since each of them involve design tradeoffs:

1. Creation of a transistor circuit topology that provides a certain digital function.

2. Sizing and ordering transistors in the circuit topology.

3. Placing routing and compacting the above transistors into a layout.

2.1 Our Research Goal

Given the ambitious goal of automating phases 1, 2 and 3 above into a single phase, we set out to explore the possibility of accomplishing this for a very simple standard cell — the CMOS inverter (Figure 4.9) — without a schematic and without incorporating any design heuristics for transistor sizing, splitting, ordering, placement, routing or compaction. The tools we decided to work with were MAGIC [18] — a layout schematic editor, SPICE [18, 19] a circuit simulator and a modified version of GADO [3]— a design optimization engine (Section 4.2.1). Our goal was to start from a behavioral description for an inverter, an optimization criterion and a set of building blocks (different types/clusters of transistors, piece of "poly", piece of "metal1", piece of "ndcontact" etc.) and attempt to automate the process of finding a working inverter optimized for the given criterion as a proof of concept. The behavioral description consisted of a truth table of inputs and outputs with all possible input transitions (0 to 1 and 1 to 0). This is because certain designs may fail on some transitions. Table 1 lists all the input transitions used for the inverter with corresponding outputs.

Table 2.1: Truth Table with all Possible Transitions for an Inverter

Input	**Output**
0	1
1	0
0	1

But what about testing different input sequences? There are an infinite number of input sequences and it is possible that a design with unnecessary states fails at a certain input sequence. Our own experience with this algorithm is that it tends to find the simplest design and hence the correct design. This is because complicated designs that have more components also occupy more area and hence

receive higher penalties than simple designs. (The penalty scheme is discussed in 4.2.3). Another question that needs to be addressed is the justification for using a behavioral description as opposed to the traditional schematic description. We believe that this added degree of freedom creates more room for innovation. The algorithm simultaneously searches for the optimal layout as well as the optimal schematic.

Chapter 3

VLSI Standard Cell Design Using Genetic Algorithms

3.1 Introduction

Standard cells are the lowest-level building blocks in VLSI design. Since most chip designs are composed of many standard cells, optimization of these cells is very important. Traditionally, standard cell libraries have been designed by humans. Recently, however, there has been a movement toward automated standard cell design [1, 2].

The advantages of automated standard cell design are low cost and custom tailoring. On the other hand, current automatically generated libraries are less efficient than hand designed ones.

3.2 Our work

Our work addresses the problem of inefficiency in automated standard cell design. We apply a genetic algorithm [3] to find standard cells, which are more efficient than hand designed ones. Genetic algorithms (GAs) are used to find more fit solutions from a diverse pool in a process similar to natural selection. The GA used in this work (GADO) is a GA that was designed to be suitable for engineering design optimization domains. These domains usually involve expensive fitness evaluations and search spaces that are very difficult to search.

3.2.1 Representation

We used integers to represent the different components on the chip. We use a repair method to reduce the effects of aliasing. When genetic operators such as crossover and mutation are done, we sort the components of the chromosome in a way that maps the order of components on the board.

3.2.2 Fitness

A number of metrics were taken into account in our fitness function. These included area, power, speed, number of correct outputs, and design rule errors. The fitness function was hierarchical so that time would not be wasted evaluating incorrect designs for efficiency. The fitness function was also extended to check that inputs influence the proper output nodes.

3.2.3 Diversity

The distance between two designs was computed as the Euclidean distance between their integer representations. We are planning to create a more meaningful diversity function by taking into account the circuit structure.

3.3 Preliminary Results

Figure 3.1 shows the current results of our work. This is an attempt to build an inverter. You can see that the input node drives the gate of a transistor influencing the output node. Unfortunately, the other node of the transistor is not connected because the fitness function did not encourage this. We are planning to extend the fitness function to ensure that no transistor nodes are unconnected.

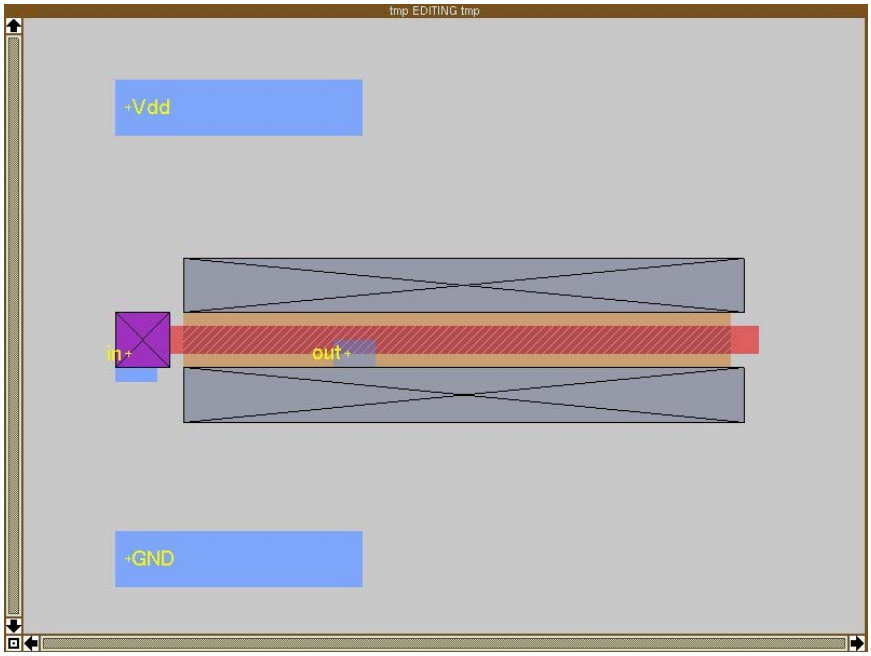

Figure 3.1: Current attempt at an Inverter.

3.4 Conclusions

We have presented a novel scheme for automated standard cell generation. We currently have not generated any complete circuits. However, we believe that this technique shows great promise with further development.

3.5 References

[1] C. Edwards, *EDA Vendors Rethink Standard-Cell Libraries*, Electronics Times, June 2000.

[2] D. Pietromonaco, *Automating Cost-Effective Library Creation*, Integrated System Design, November 2000.

[3] K. Rasheed and H. Hirsh, *Learning to be Selective in Genetic-Algorithm-Based Design Optimization*, Artificial Intelligence in Engineering, Design, Analysis

and Manufacturing, 1999.

Chapter 4

An Evolutionary Approach to Standard Cell Synthesis from Behavioral Descriptions: A proof of concept

4.1 Introduction

Standard cell methodology is widely used in IC design. Considerable effort has been invested in attempting to automate the design of a standard cell. Automation of standard cell layout generation (1) significantly improves turnaround time for creating new standard cell libraries, (2) provides a test bed for evaluating new process technologies by rapidly synthesizing cells and (3) enables rapid migration of designs to new process technologies. Most methods for cell synthesis are schematic-based methods as they start with a sized/unsized transistor-level netlists and automate the processes of placement, followed by routing and possibly compaction. Our implementation attempts to build a cell on-the-fly given only a behavioral description. The input specification consists of:

1. A truth table description of the input and output with as many possible transitions (more on this in 4.1.2).

2. A set of building blocks consisting of geometries of objects such as transistors, piece of polysilicon, piece of polycontact etc.

3. A classification of labels used (as inputs and outputs).

4. A cell template specifying the placement of fixed ports (if any).

A design rule checker (MAGIC) and circuit simulator (SPICE) are also used by the Genetic Algorithm to evaluate and validate designs.

4.1.1 Motivation

As noted earlier there is a progression toward more automated cell synthesis techniques. Some authors [5, 6] advocate a shift in the use of standard cell libraries, as we know them. The idea here is to engineer standard cells on demand or "on-the-fly" thus eliminating the need for developing and maintaining expensive static cell libraries. This would allow the designer to demand cells that may not exist in any pre-defined library, optimized for a given criterion depending on the context. In particular this would permit [10]:

1. Standard cell and datapath placement and routing tools to request cells with exact pin-orderings.
2. Logic synthesis tools to request specific logic decompositions.
3. Interconnect optimization tools to request cells with specific input and output impedance values.
4. Power optimization tools to request specific power/delay trade-offs, perhaps even specify a logic family.

Other advantages include:

1. Synthesis of cells of desired shape and size.
2. Transistor level tuning.
3. Ability to jointly conduct circuit and process optimization across process and layout architecture spaces [9].

Typical techniques are schematic dependent. They either start from a netlist of transistors with the transistor type and sizing specified by the designer or from some other type of a symbolic layout followed by compaction. Existing compaction techniques are unable to change the shape or orientation of layout objects and thus depend on the designer's specification. Yet another technique used is the development of procedural module generators that are programs that hard code procedures for assembling a cell as well as design rule constraints. Hard coding limits their use in the face of changes in cell architecture and interconnect technology [5].

Fixed libraries make device level tuning impossible (or difficult). Poor timing, area or power tradeoffs may result since the design of the cell is decoupled from the constraints imposed by higher-level tools. Schematic independence and device-size tuning is best accomplished via on-the-fly leaf cell synthesis. The use of C5M to develop a 440 MHz processor for IBM discussed in [6] is a good example of such a design methodology. Lefebvre, Marple and Sechen [5] note that layout synthesis tools need to optimize across all of the following phases since each of them involve design tradeoffs:

1. Creation of a transistor circuit topology that provides a certain digital function.
2. Sizing and ordering transistors in the circuit topology.
3. Placing routing and compacting the above transistors into a layout.

4.1.2 Our Research

Given the ambitious goal of automating phases 1, 2 and 3 above into a single phase, we set out to explore the possibility of accomplishing this for a very simple standard cell — the CMOS inverter — without a schematic and without incorporating any design heuristics for transistor sizing, splitting, ordering, placement, routing or compaction. The tools we decided to work with were MAGIC [14] — a layout schematic

editor, SPICE [14, 15] a circuit simulator and a modified version of GADO [3]—a design optimization engine (Section 4.2.1). Our goal was to start from a behavioral description for an inverter (Figure 4.1), an optimization criterion and a set of building blocks (different types/clusters of transistors, piece of "poly", piece of "metal1", piece of "ndcontact" etc. as seen in Figure 4.4) and attempt to automate the process of finding a working inverter optimized for the given criterion as a proof of concept. The behavioral description consisted of a truth table of inputs and outputs with all possible input transitions (0 to 1 and 1 to 0). This is because certain designs may fail on some transitions. Figure 4.1 lists all the input transitions used for the inverter with corresponding outputs.

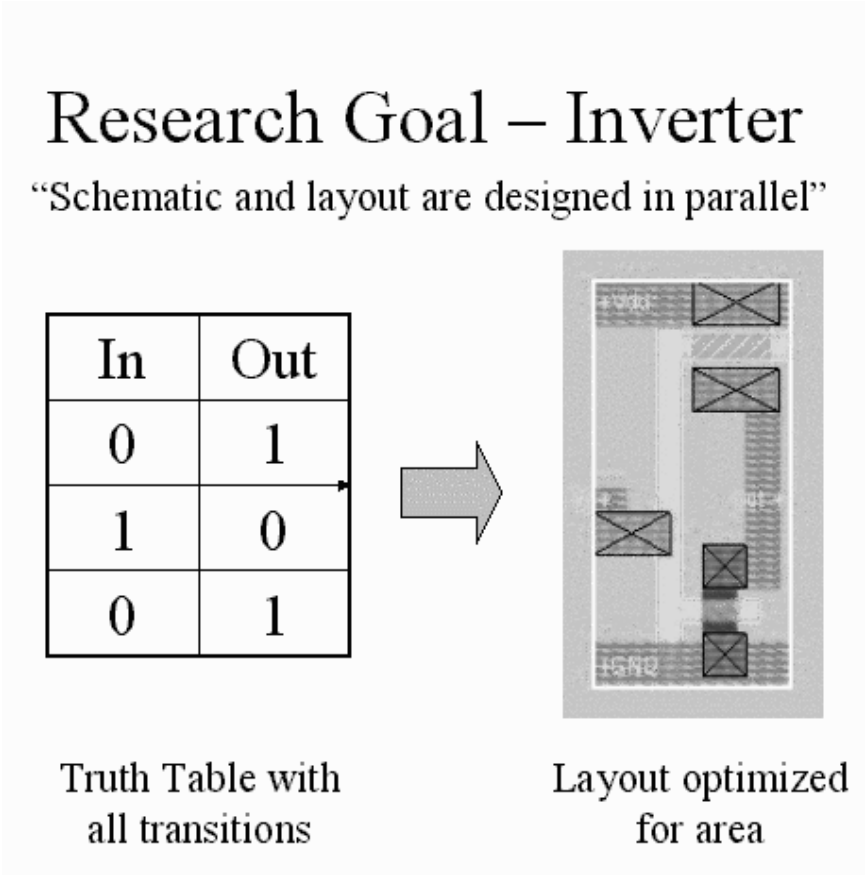

Figure 4.1: Research Goal

But what about testing different input sequences? There are an infinite number of input sequences and it is possible that a design with unnecessary states fails at a certain input sequence. Our own experience with this algorithm is that it tends to find the simplest design and hence the correct design. This is because complicated designs that have more components also occupy more area and hence receive higher penalties than simple designs. (The penalty scheme is discussed in 4.2.3). Another question that needs to be addressed is the justification for using a behavioral description as opposed to the traditional schematic description. We believe that this added degree of freedom creates more room for innovation. The algorithm simultaneously searches for the optimal layout as well as the optimal schematic.

4.2 An Evolutionary Approach

4.2.1 Genetic Algorithms for Design Optimization

Genetic Algorithms (GAs) are a class of heuristic algorithms that can search for a solution by a process similar to natural selection, i.e. the desired characteristics of the solution are enhanced by continued breeding. We use GADO, which is a steady state Genetic Algorithm for Design Optimization tool developed by Khaled Rasheed for his PhD dissertation [3], for applications in design optimization including aerospace industry applications [7], where coupled large-scaled optimization problems commonly arise. GADO maintains a population of potential designs (standard cell designs in our case). Better designs are generated using Crossover and Mutation operators within the algorithm. Since designs get better and better with time such a design methodology is also known as design by evolution. Thus the design of the schematic and the optimization of the layout happen in parallel at both log-

ical (schematic/connectivity) and physical (layout) levels as opposed to traditional methods.

Figure 4.2 shows the details of our architecture. The designs were generated by GADO and were evaluated by a fitness function. The fitness function would invoke the design rule checker in MAGIC and SPICE when additional evaluations needed to be done (as described in Section 4.2.3).

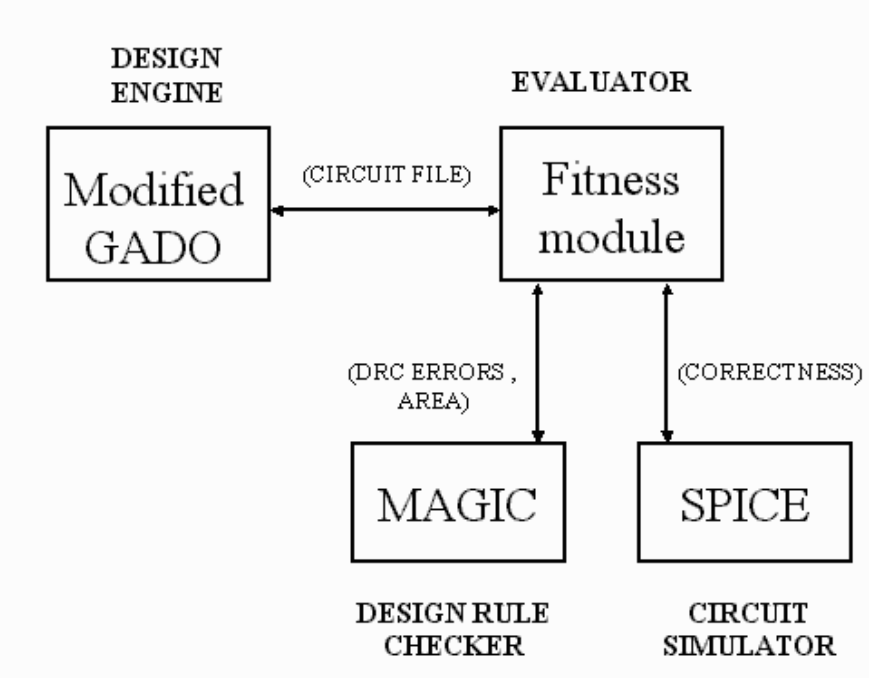

Figure 4.2: Architecture

4.2.2 Coding Scheme

The coding scheme refers to the representation of a cell. We use integers to encode a layout. Each design is represented in the algorithm as a string (Figure 4.3) of Objects corresponding to each component in the physical layout. An object is a member of a pre-defined set of building blocks including various types of transistors,

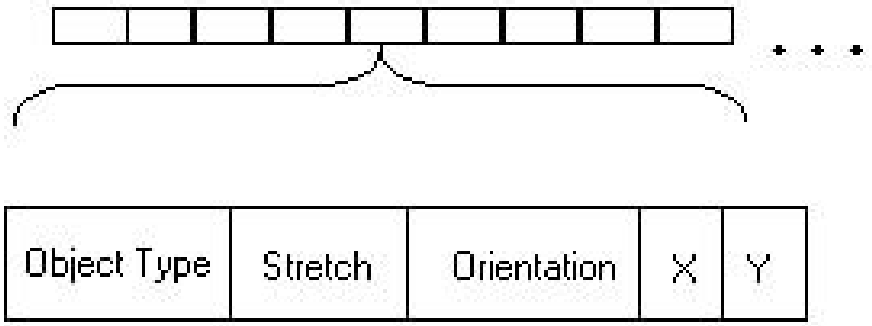

Figure 4.3: Representation of a Standard Cell Layout

piece of polysilicon, piece of polycontact etc. Figure 4.4 is a snapshot of a MAGIC file with all the blocks placed in a single file.

It may be noted that there are 15 object types that consist of symmetrical and asymmetrical transistors of different types (single, dual and triple). Three other blocks are pieces of polycontact, polysilicon, and "metal1" (extreme right, top to bottom in Figure 4.4). The user may easily add other building blocks as and when required. Each object is further defined by its type, X and Y coordinates, stretch and orientation (Figure 4.3). Cell limits refer to the size of the grid of the maximum allowable cell size (as defined by the user). This forces the GA to place components only within the grid. The orientation refers to a number (0, 1, 2 or 3) that decides the rotation applied to a block (0, 180, 90 or 270 degrees respectively). The stretch factor is a number corresponding to the number of units by which a block is scaled. The scaling is evenly distributed along the direction of orientation. The maximum scale is limited by the maximum allowable cell size. X and Y refer to the coordinates of the center of the object and are thus also constrained to be within the maximum allowable cell area.

Table 4.2.2 summarizes the parameters of each object (column 1), the range of values that they may have (column 2) and illustrates the values corresponding to

Table 4.1: Ranges of Parameters and Examples of 2 cells

Parameters	**Values**	**Figure 4.5**	**Figure 4.6**
Object Type	1-15	11	11
Orientation	0-3	0	2
Stretch Factor	Cell Limits	0	5
X	Cell Limits	0	15
Y	Cell Limits	0	14

the objects shown in Figures 4.5 and 4.6. Note the effects of moving, stretching and rotating the same object. Figures 4.5 and 4.6 also demonstrate the use of templates. The 3 pieces of metal and the 4 labels (Vdd, Gnd, in and out) are part of a user defined template over which the other objects are superimposed.

The objects in the individual were sorted by Euclidean distance from the origin to enable meaningful crossover.

4.2.3 Smart Fitness Function

The fitness function (also known as *cost function*) checks a candidate design against a list of constraints and penalizes the design for every violation encountered. The penalty values vary for each constraint depending on their importance. This way the GA fixes the most serious violations first and progresses to make smaller fixes. The checks involving the external simulators (MAGIC and SPICE as seen in Figure 4.2) are only made after a minimum set of constraints are satisfied. This way time is not wasted on checking circuits that are obviously incorrect (such as those with overlapping transistors). Only a circuit with fully connected transistors and labels go through a SPICE simulation. Along a similar vein, time is not wasted checking incorrect circuits for efficiency.

In the first phase the GA tries to weed out designs with overlapping transistors. After succeeding in that venture it goes on to check connectivity of individual

transistors and tries to encourage connections (more details in Section 4.3). Simultaneously, the design rule checker is invoked so that the GA tries to minimize both design rule violations as well as broken connections.

After the GA has succeeded in creating designs with no broken connections it passes on to the next phase where the design rule check is made in conjunction with a circuit simulation. In this stage, the GA is trying to simultaneously satisfy the requirements of zero design errors and zero simulation errors. Once the GA has come up with a working circuit it proceeds to optimize the design for a given criterion (i.e. search in the space of correct designs), which is usually one of the following:

1. Minimum Area
2. Minimum Delay times
3. Minimum Power dissipation

One of the most challenging aspects of the project was developing, what was described as the *transistor connectivity* check component. A directed graph was used to capture connectivity information. To illustrate the concept of nodes and connections here is an example of a circuit layout (Figure 4.7) followed by the graph (Figure 4.8) used to capture connectivity information.

The transistor connectivity check is also called the *influence check* since its function is to penalize nodes that do not influence nodes that they ought to. We also need to formally define *influence* (arrows in the graph).

Node A influences node B if there is

1. A direct electrical connection, or
2. A unidirectional electrical influence such as the control of the gate over the source and drain of a transistor.

In Figure 4.7, nodes 1 and 2 influence one another by property (1); the bidirectional nature of influence in represented by solid lines. Also nodes 3 and 4 exhibit property (2); an arrow going from node 4 to node 3 represents the unidirectional nature of influence.

It may be also noted that all labels were classified as inputs or outputs. Vdd and Gnd were treated as inputs. The rules that the influence check tests are:

1. None of the labels should be shorted.
2. Every input must influence at least one output.
3. Every output must be influenced by at least one input.
4. Every transistor gate must be influenced by at least one input.
5. Both source/drain terminals of a transistor must be influenced by at least one input or influence at least one output.

Thus the layout in Figure 4.7 violates rules 2 and 4 above as may be seen in the graphical schematic in Figure 4.8. Let us consider another case where there is a violation, say in rule 3 above (not shown in figures). It is more useful to know how bad the violation is than a binary yes/no. This is accomplished by a distance check function that in case of a violation in rule 3 will do a breadth-first graph traversal starting from the floating output to determine all the nodes that may influence it creating an O-list. Similarly another graph traversal is made from each input to create their respective I-list of nodes that they influence. Once created, the distance function finds the shortest Manhattan distance between nodes in the I-lists and the node in the O-list and returns a penalty based on this distance. Note that many of these nodes may be in different layers. We use a look up table with values preset by a domain expert to calculate distances between nodes in different layers.

Since designs in which a node in the I-lists is closer to a node in the O-lists receive lower penalties, the GA favors these designs and in conjunction with the other 4 rules above it has the holistic effect of encouraging connections between inputs and outputs via transistors without shorting inputs. This idea lies at the heart of the GA's success. To summarize:

If you have some label that is not being influenced by any other label, we want to know how close it is to some label that can influence it.

The idea that information about the degree of the violation is more useful than just knowing whether there is a violation is another key ingredient to the GA's success and is used in the next 2 stages too. In the design rule check stage, a script is fed to MAGIC in the NULL mode and the number of tiles that are involved in the design rule errors is summed to arrive at a penalty. In the circuit simulation step, a SPICE simulation is made and a penalty is arrived at by dividing the total number of entries in the truth table simulated incorrectly divided by the total number of entries in the truth table.

4.2.4 Smart Distance Function

A distance function takes 2 designs as arguments and returns a measure of "how different" they are. This is used by a *diversity module* within GADO to weed out duplicates and designs that are very similar in order to encourage diversity in the population of potential designs. This helps the GA maintain a much more thorough representation of the search space as well as allowing it to escape from local optima.

In the earlier runs, while the fitness function was under construction a simple distance function based on the Euclidean distance between 2 strings was used. An example with 2 hypothetical simple strings of 4 *alleles* (numbers in the string) each (say A and B) illustrates the idea:

```
Individual A :  11  3  0  1
Individual B :   9  3  1  0
```

The 4-dimensional Euclidean distance is calculated as the square root of

$$\sum_{i=1}^{4} (A_i - B_i)^2$$

$$= \sqrt{(11-9)^2 + (3-3)^2 + (0-1)^2 + (1-0)^2}$$
$$= \sqrt{6}$$

The problem with this simple distance function is that it is not sensitive to the objects in the design since distance between each allele is calculated on the same scale. After preliminary success with the GA, we set out to fix this. The idea was to use a weighted scheme corresponding to what each allele represented and how distinguishing a feature it was in the context of the entire circuit.

In the above example the distance might be:

$Sum = A\ |11-9| + B\ |3-3| + C\ |0-1| + D\ |1-0|$

where *A, B, C* and *D* are weights that reflect the importance of each allele in its contribution to the total distance between the given designs. The `||` brackets here represent the absolute difference. In practice, for a cell with 6 objects the number of alleles is 30 (since each object is coded with 5 parameters as shown in Figure 4.3). We ended up using this scheme with an additional step explained in the next paragraph.

Before the final distance between the 2 individuals (say S1 and S2) was computed, every object in S1 was compared with every object in S2. The objects were rearranged within the individuals so that of all possible distances between the individuals, the minimum distance was computed. This rearranging took place locally within the distance computation function so that the actual layouts were not modified.

Our scheme still does not fix the problem of *apparent diversity*. There are many physical realizations of a given transistor schematic each of which may have its own unique sequence of alleles. The function needs to be smart enough to say that these designs are more similar than designs that may have smaller distances but represent a totally different schematic. This is because in this domain, small changes in the location, size or orientation of the objects can cause large changes in their logical equivalents. Such a function would have to be *schematic sensitive* by computing the distance between the graphs of the schematics. We are now investigating this and also a different representation scheme [8] that can make distance computations easier.

4.3 Experimental Results

These are the results from our attempts to build an inverter. Figures 4.9, 4.10 and 4.11 show MAGIC layout editor screen shots of the inverters designed by the GA for three different templates with different label configurations. The complete snapshots from the evolution of these circuits may be viewed at our website [17]. The process we followed to test these templates was to:

1. Plan a template,
2. Hand design the inverter using the template and
3. Modify the template if necessary to eliminate design rule errors.

Once we were confident that a hand-crafted solution existed for a given template, the GA was designated the task of finding it. Figures 4.12-4.19 illustrate the highlights in the evolution of the design of the inverter shown in Figure 4.9.

The C code ran on a Linux box over several iterations. As the GA progressed it placed the designs that were "best so far" (individuals that were evaluated with

the highest fitness) on our web page [17] so that we could monitor its progress. The screenshots you see here are some of the highlights in the evolution of the cell. The run lasted 3 hours (the next section discusses how this may be reduced) on a 1.2 GHz machine. The first snapshot (Figure 4.12) is that of a random placement of 6 (or as defined by user) objects from the building-block library (Figure 4.4). Subsequent snapshots show the improvement of the circuit over time. The GA first weeds out overlapping transistors seen in 4.13 and arrives at a simple design with a few objects and a lot of broken connections (Figure 4.14). It then attempts to fix these connections by reducing the distance between nodes that could influence one another (Figures 4.15-4.17). On succeeding to fix all connections (Figure 4.18) it attacks the final hurdle of coming up with a working circuit with zero design rule errors (Figure 4.19). It then goes on to minimize the area (user defined criterion) occupied by this circuit and arrives at the final design shown in Figure 4.9.

4.4 Limitations

The GA can currently design cells with a small number of transistors. The search space becomes intractable as more and more objects are added. Plenty of optimizations can also be made within GADO. A better encoding [8] of the design (representation of a layout within the GA) is needed to tackle the combinatorial challenges of cells such as a full adder. As may be seen at our website [17], all runs of the GA do not converge at the best design. Different random seeds that the GA started with would converge at different designs. This means that the GA does not always find the global optima.

4.5 Conclusions

An evolutionary approach (using GADO, a genetic algorithm) for standard cell design automation was proposed. GADO explores the space of all possible configurations (of a set of building blocks) given only a behavioral description of the circuit. The search space includes all possible electrical connectivity and layout and is accomplished in a reasonable time frame for an inverter (3 hours on a single processor). The same result may be obtained in a fraction of the time by adding multiple processors since genetic algorithms are easily implemented on parallel architectures [4] or a network of workstations. Statistical observations show that for any final desired result achievable using the serial algorithm, the parallel version could provide almost linear speedup (i.e. N times faster on N processors) [13, 11].

Thus the *design* and *optimization* of the inverter is done in parallel at both logical (schematic) and physical (layout) levels. A working inverter was designed as a proof of concept. This approach has the flexibility of generating cells on the fly to address the ad-hoc constraints faced by the designer when she is considering a candidate design in a larger context such as choosing a cell. This was demonstrated by the design of inverters with arbitrary label placements. This allows higher-level standard cell and datapath placement and routing tools to request cells with exact pin-orderings.

A smarter distance function and alternative representations are being investigated in an attempt to make the search space more tractable for more complex cells such as a full adder.

4.6 References

[1] R. K. Brayton, G. D. Hechtel and A. L. Sangiovanni-Vincentelli. "A Survey of Optimization Techniques for Integrated Circuit Design", *Proceedings of the*

IEEE, Vol. 69, No. 10, 1981 pp. 1334-1363.

[2] B. Bishop, K. Rasheed and A. Bahuman. "VLSI Standard Cell Design Using Genetic Algorithms", *39th Annual ACM Southeast Conference*, 2001.

[3] K. Rasheed. "GADO: A Genetic Algorithm for Continuous Design Optimization", PhD. Thesis. http://www.cs.uga.edu/ khaled

[4] E. D. Goodman. "An Introduction to GALOPPS the Genetic Algorithm Optimized for Portability and Parallelism System", *CASE Center Technical Report* No. 940401, Michigan State University, 1994, pp. 58.

[5] M. Lefebvre, D. Marple and C. Sechen. "The Future of Custom Cell Generation in Physical Synthesis", *IEEE 34th Design Automation Conference*, 1997, pp. 446-451.

[6] J. Burns and J. Feldman, "C5M A Control Logic Layout Synthesis System for High-Performance Microprocessors", *IEEE Trans. On CAD*, 17(1), 1998, pp. 14-23.

[7] G. Carrier, D. Knight, K. Rasheed and X. Montazel. "Multi-criteria Design Optimization of a Two dimensional Supersonic Inlet", *39th AIAA Aerospace Sciences Meeting and Exhibit*, 2001, AIAA Paper 96-4142 pp.1355.

[8] H. Murata, K. Fujiyoushi, S. Nakatake, and Y. Kajitani, "Rectangle packing based module placement", *Proceedings IEEE International Conference on Computer-Aided Design*, 1995,pp. 472–479.

[9] D. G. Baltus, T. Varga and R. C. Armstrong, "Developing a concurrent methodology for standard-cell library generation", *IEEE 34th Design Automation Conference*, 1997, pp. 333-336.

[10] M. A. Riepe, K. A. Sakallah, "Transistor Level Micro-Placement and Routing for Two-Dimensional Digital VLSI Cell Synthesis", *International Symposium on Physical Design*, April 12-14, 1999, pp. 74-81.

[11] K. Rasheed, B. Davison, "Effect of Global Parallelism on the Behavior of a Steady State Genetic Algorithm for Design Optimization", *Congress on Evolutionary Computation*, 1999, pp. 534–541.

[12] T. Lengauer. "Combinatorial Algorithms for Integrated Circuit Layout", Chichester, England: Wiley 1990.

[13] P. Mazumder and E. M. Rudnick, "Genetic Algorithms for VLSI Design, Layout & Test Automation", Prentice Hall, 1999, pp. 264-265.

[14] http://www.research.compaq.com/wrl/projects/magic/

[15] http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/

[16] http://ece.www.colorado.edu/ ecen4228/spice/spice.htm

[17] http://james.cs.uga.edu

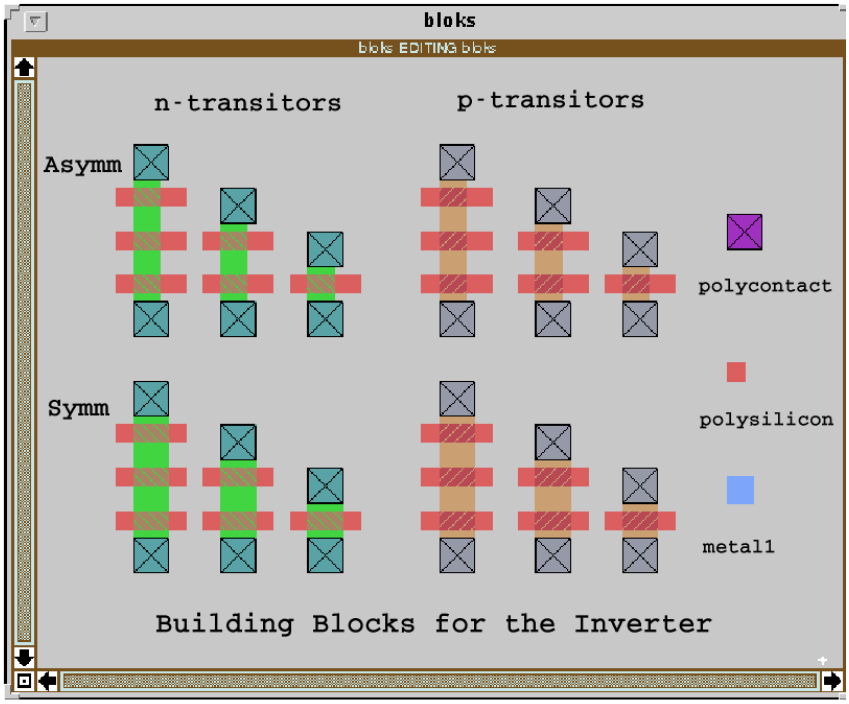

Figure 4.4: Different types of Objects used in the construction of an Inverter

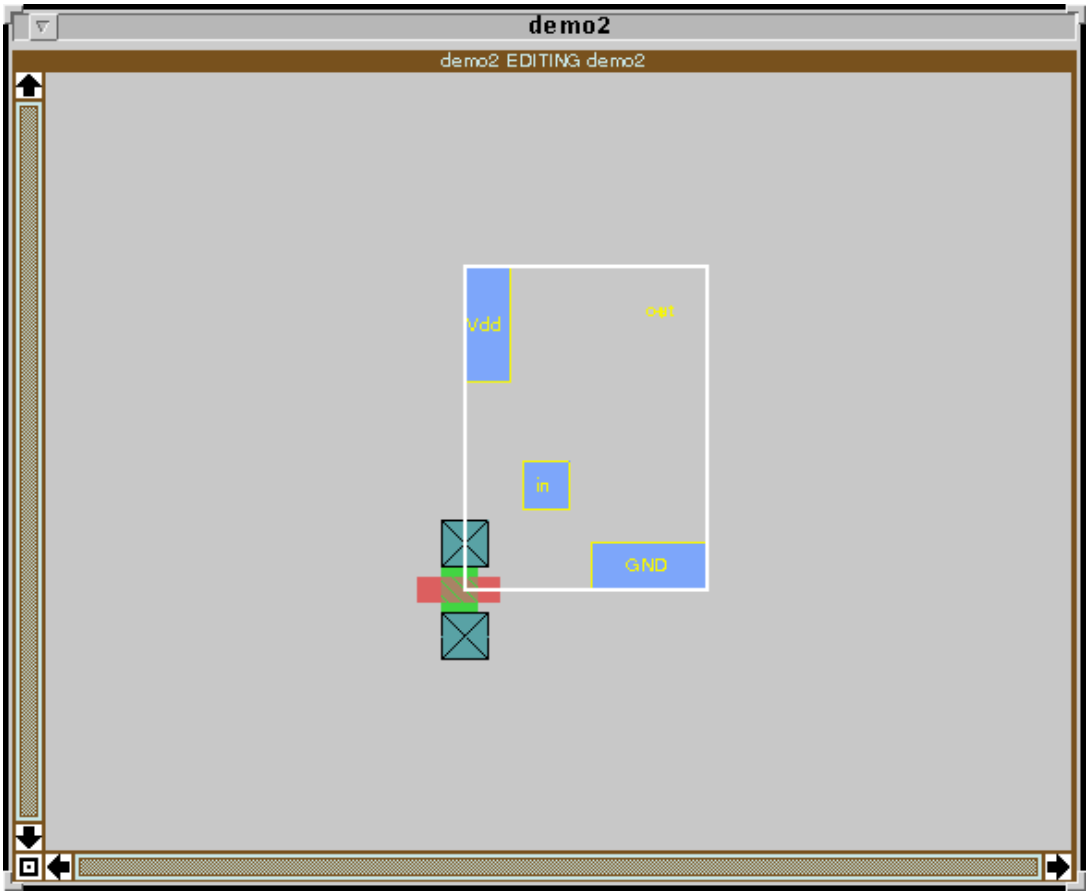

Figure 4.5: A cell with a template and a single Object

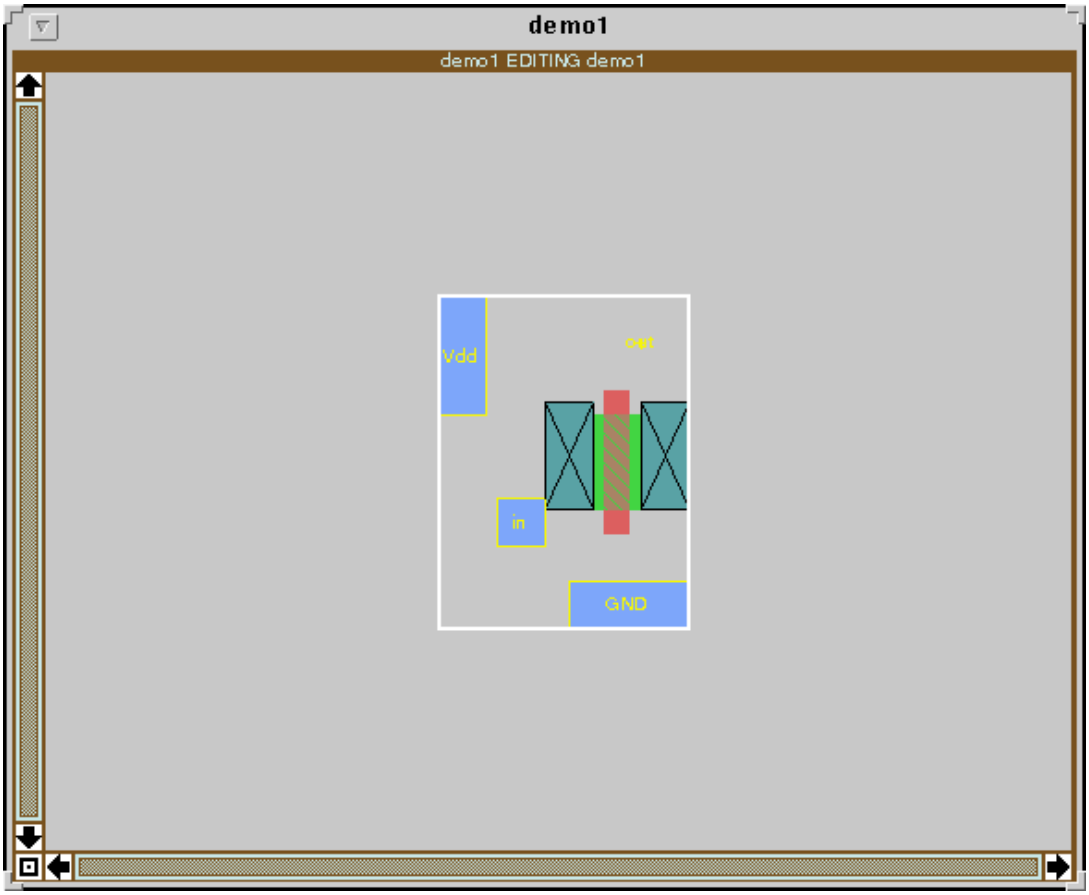

Figure 4.6: The same cell with the Object moved, stretched and rotated

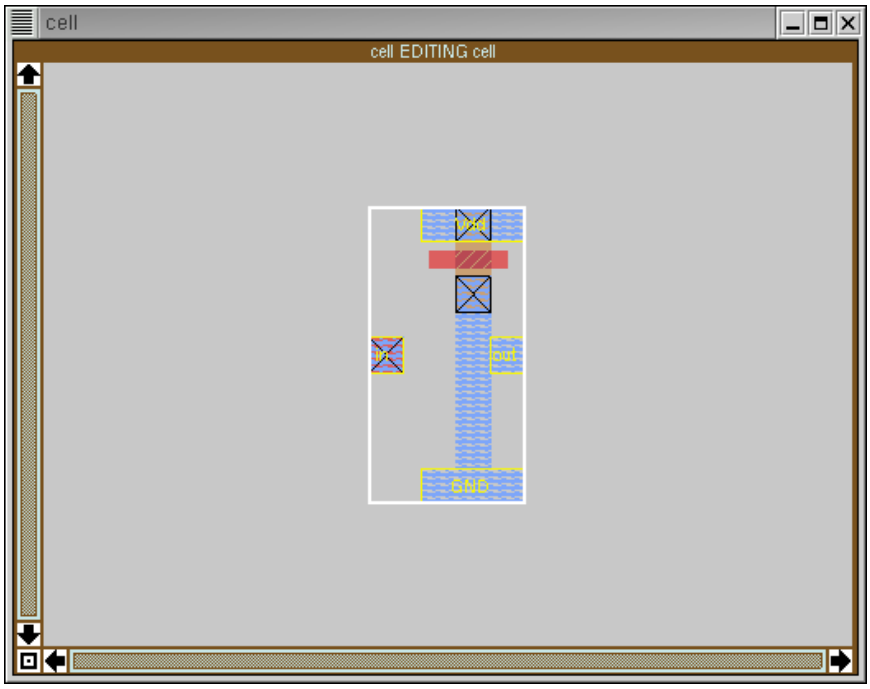

Figure 4.7: A Sample cell

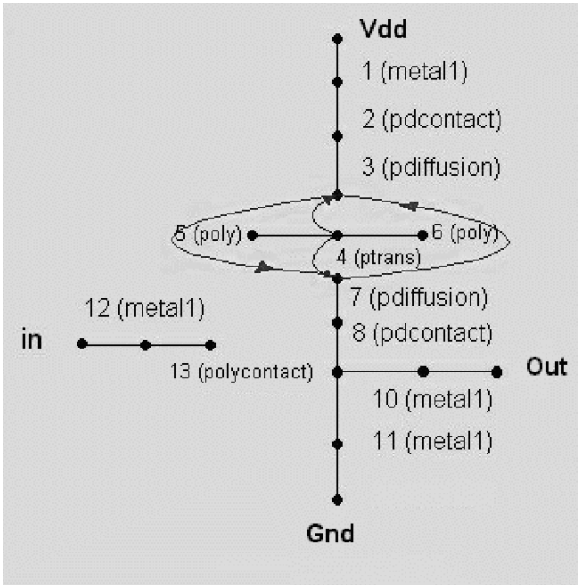

Figure 4.8: Connectivity Graph corresponding to cell in Figure 4.7

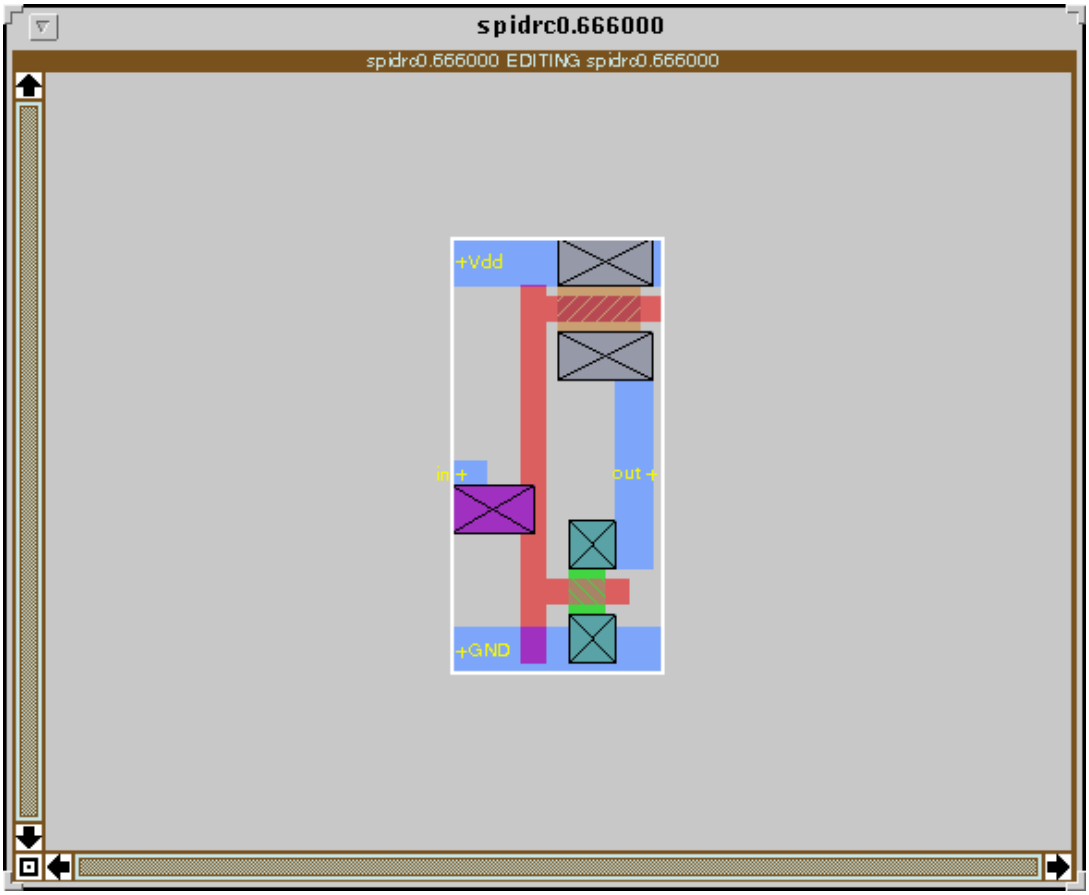

Figure 4.9: An Inverter

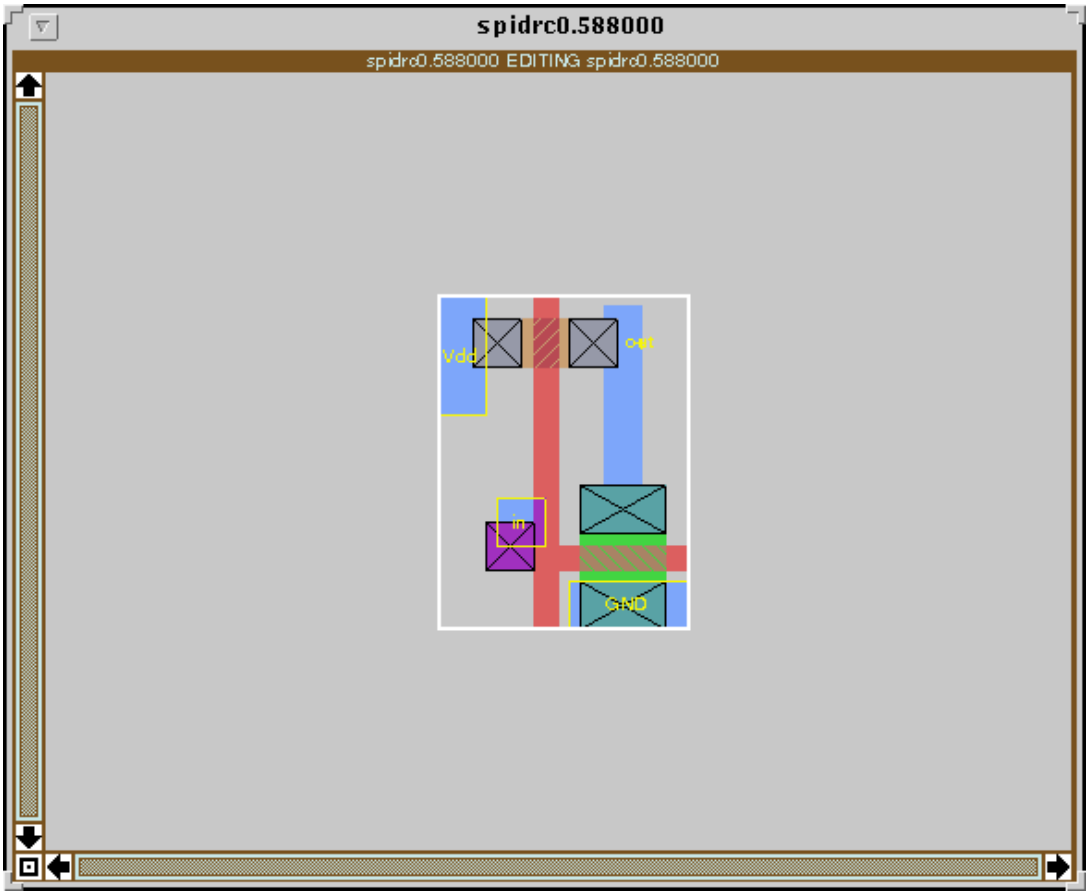

Figure 4.10: An Inverter with arbitrary label placement (1)

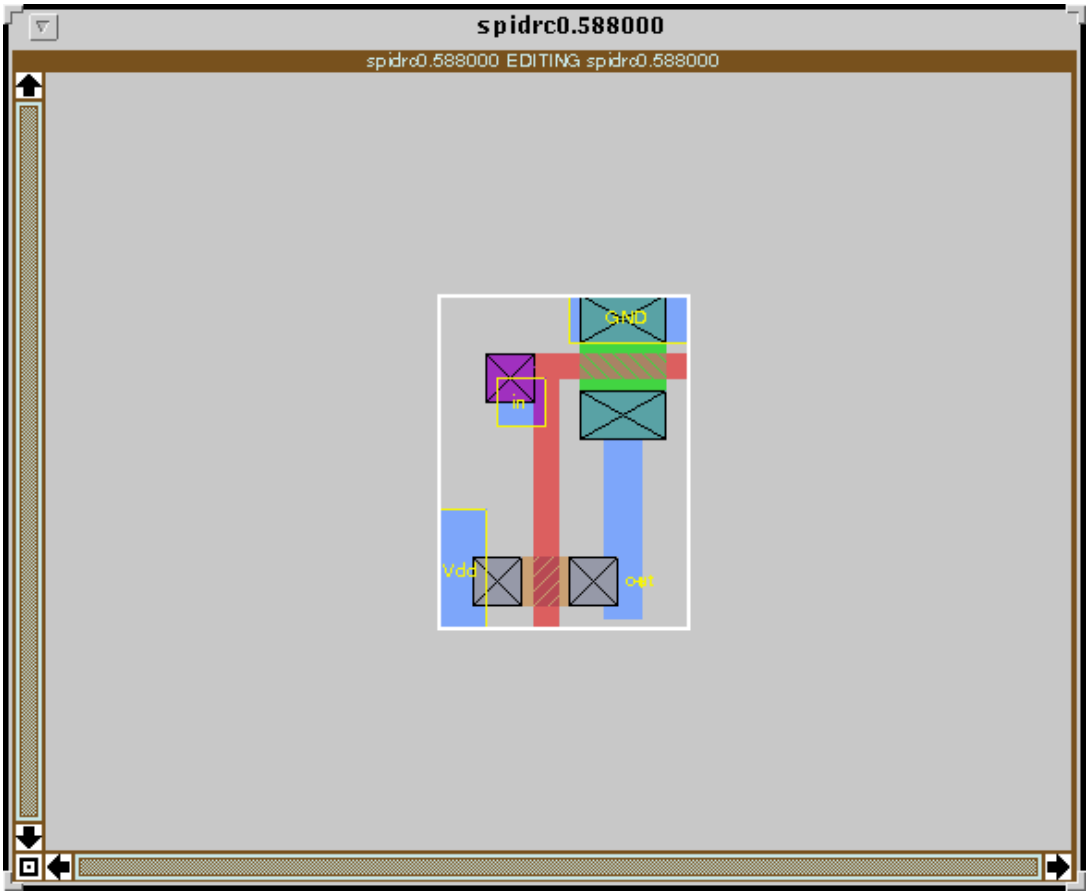

Figure 4.11: An Inverter with arbitrary label placement (2)

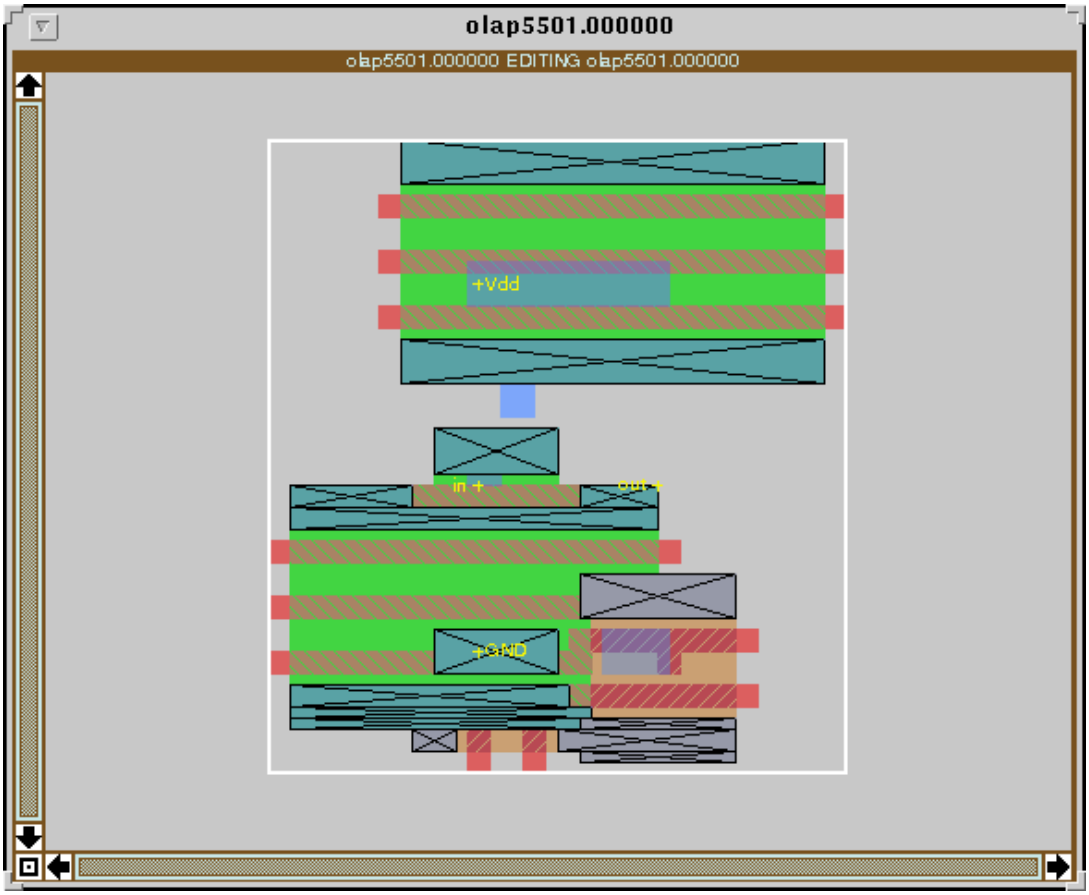

Figure 4.12: First attempt at the Inverter

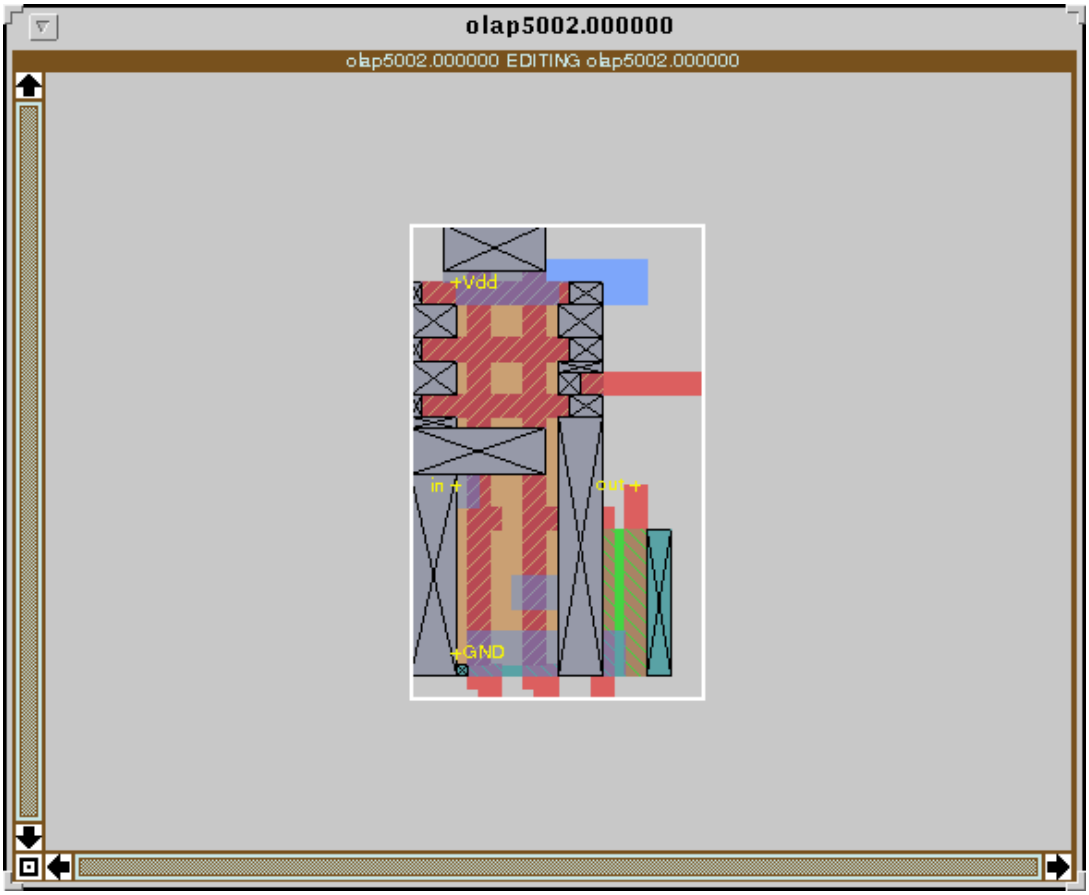

Figure 4.13: Overlapping Transistors

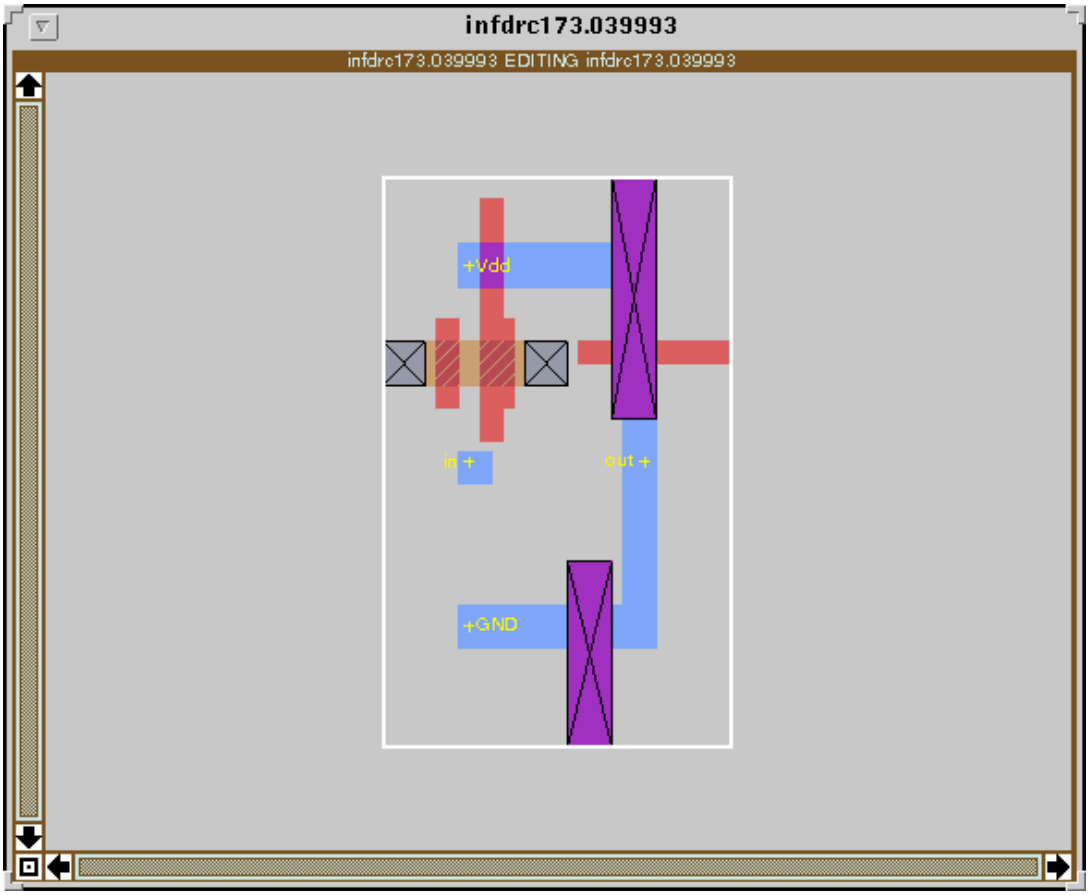

Figure 4.14: A simple design

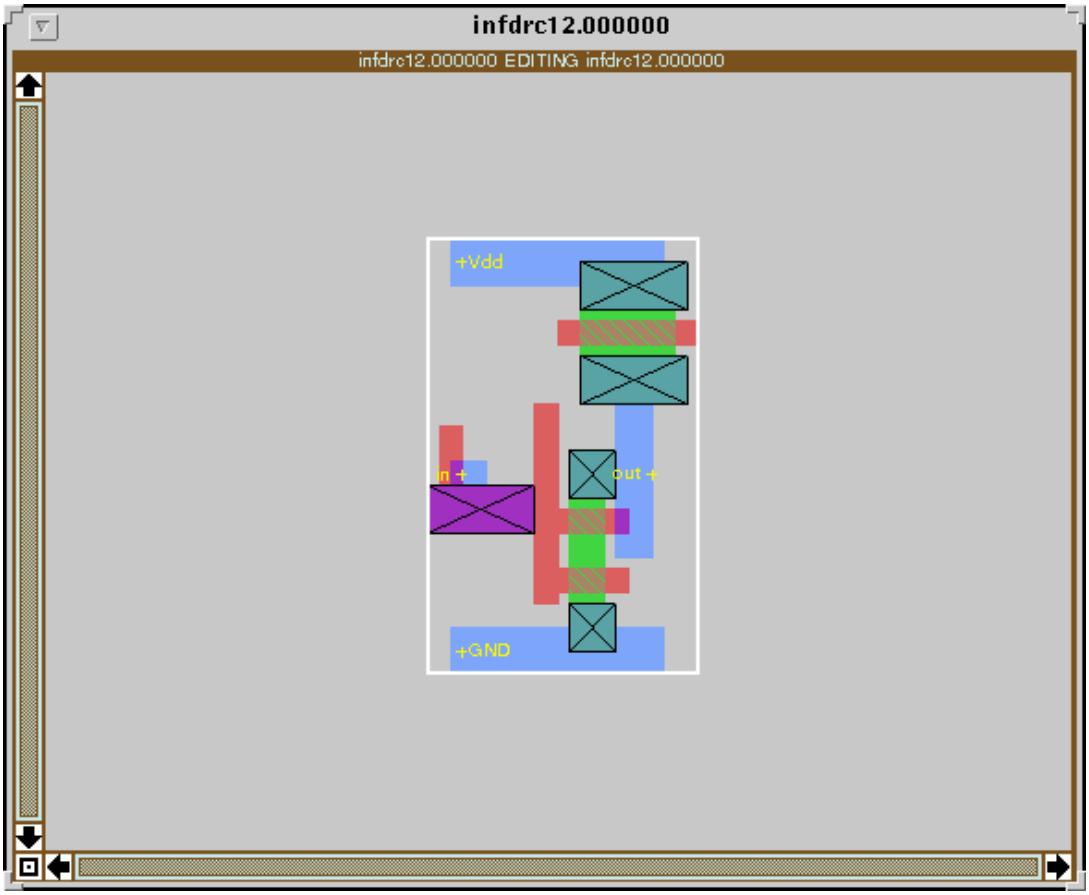

Figure 4.15: Fixing a connection (1)

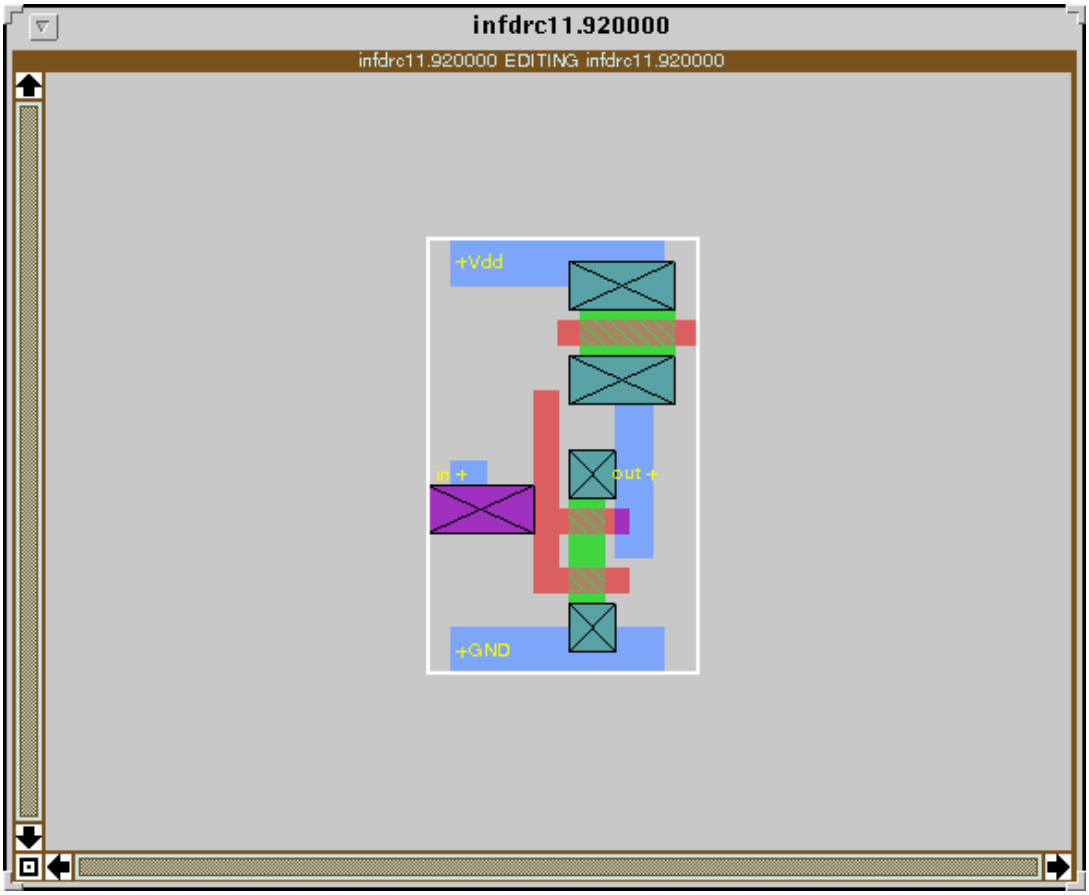

Figure 4.16: Fixing a connection (2)

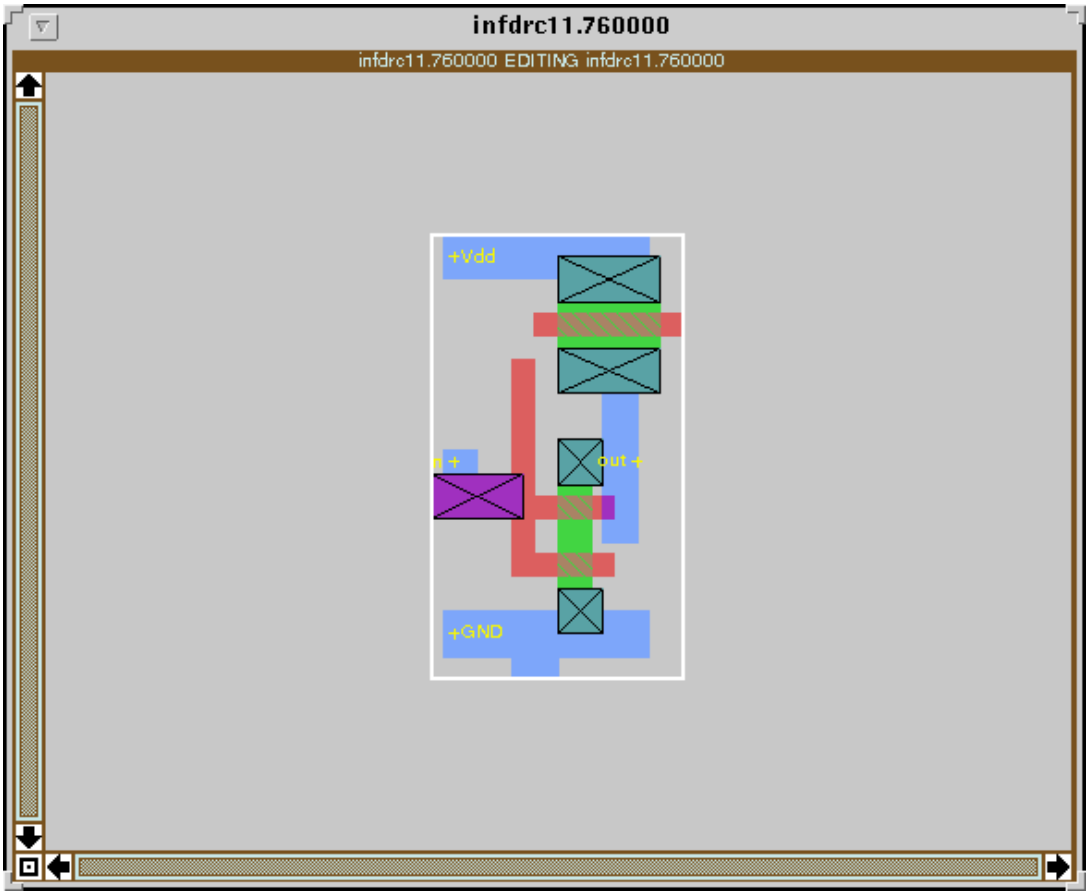

Figure 4.17: Fixing a connection (3)

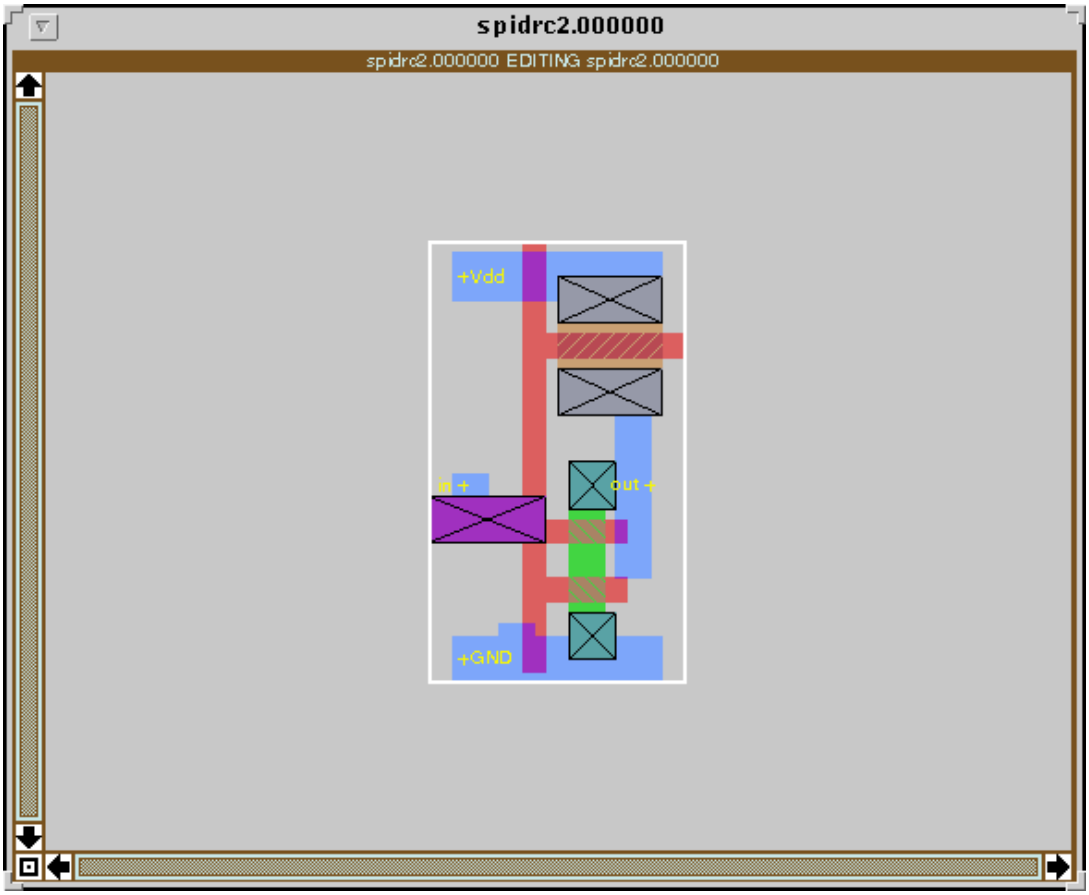

Figure 4.18: Connection is made (4)

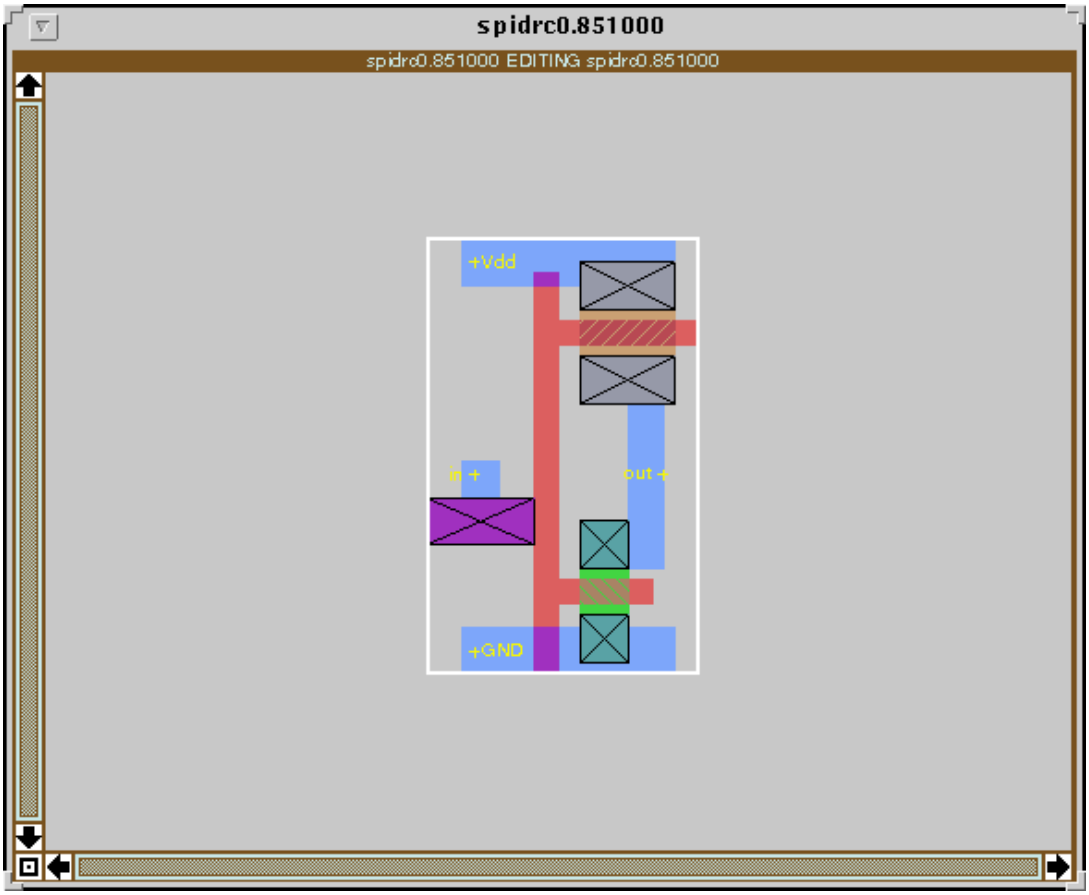

Figure 4.19: A working inverter (not yet optimized for area)

Chapter 5

Conclusions

An evolutionary approach (using GADO, a genetic algorithm) for standard cell design automation was proposed. GADO explores the space of all possible configurations (of a set of building blocks) given only a behavioral description of the circuit. The search space includes all possible electrical connectivity and layouts.

Thus the *design* and *optimization* of the inverter happen in parallel at both logical (schematic) and physical (layout) levels. A working inverter was designed as a proof of concept in 3 hours on a single processor. The same result may be obtained in a fraction of the time by adding multiple processors since genetic algorithms are easily implemented on parallel architectures [4] or a network of workstations. Statistical observations show that for any final desired result achievable using the serial algorithm, the parallel version could provide linear speedup (i.e N times faster on N processors) [12].

This approach has the flexibility of generating cells on the fly to address the ad-hoc constraints faced by the designer when she is considering a candidate design in a larger context such as choosing a cell. This was demonstrated by the design of inverters with arbitrary label placements. This allows higher-level standard cell and datapath placement and routing tools to request cells with exact pin-orderings.

A smarter distance function and alternative representations are being investigated in an attempt to make the search space more tractable for more complex cells such as a full adder.

The inverter was designed as a proof of concept. A tool that is extended to be able to design more complex cells such as a full adder would be another milestone in the field of standard cell design automation and will be of commercial interest to EDA vendors such as Prolific Inc. (Newark, CA) and Cadabra Design Automation (Santa Clara, CA) who are racing against time to develop tools that can generate standard cells on the fly [16, 17].

Bibliography

[1] R. K. Brayton, G. D. Hechtel and A. L. Sangiovanni-Vincentelli. "A Survey of Optimization Techniques for Integrated Circuit Design", *Proceedings of the IEEE*, Vol. 69, No. 10, pp. 1334-1363, 1981.

[2] B. Bishop, K. Rasheed and A. Bahuman. "VLSI Standard Cell Design Using Genetic Algorithms", *39th Annual ACM Southeast Conference*, March 2001.

[3] K. Rasheed. "GADO: A Genetic Algorithm for Continuous Design Optimization", PhD. Thesis. http://www.cs.uga.edu/ khaled

[4] E. D. Goodman. "An Introduction to GALOPPS the Genetic Algorithm Optimized for Portability and Parallelism System", *CASE Center Technical Report* No. 940401, Michigan State University, 1994, pp. 58.

[5] M. Lefebvre, D. Marple and C. Sechen. "The Future of Custom Cell Generation in Physical Synthesis", *IEEE 34th Design Automation Conference*, July 1997, pp. 446-451.

[6] J. Burns and J. Feldman, "C5M A Control Logic Layout Synthesis System for High-Performance Microprocessors", *IEEE Trans. On CAD*, 17(1), January 1998, pp. 14-23.

[7] G. Carrier, D. Knight, K. Rasheed and X. Montazel. "Multi-criteria Design Optimization of a Two dimensional Supersonic Inlet", *39th AIAA Aerospace Sciences Meeting and Exhibit*, 2001.

[8] H. Murata, K. Fujiyoushi, S. Nakatake, and Y. Kajitani, "Rectangle packing based module placement", *Proceedings IEEE International Conference on Computer-Aided Design*, pp. 472–479, 1995.

[9] D. G. Baltus, T. Varga and R. C. Armstrong, "Developing a concurrent methodology for standard-cell library generation", *IEEE 34th Design Automation Conference*, July 1997, pp. 333-336.

[10] M. A. Riepe, K. A. Sakallah, "Transistor Level Micro-Placement and Routing for Two-Dimensional Digital VLSI Cell Synthesis", *International Symposium on Physical Design*, April 12-14, 1999, pp. 74-81.

[11] T. Lengauer. "Combinatorial Algorithms for Integrated Circuit Layout", Chichester, England: Wiley 1990.

[12] P. Mazumder and E. M. Rudnick, *Genetic Algorithms for VLSI Design, Layout & Test Automation*, Prentice Hall, 1999.

[13] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, MA:Addison-Wesley, 1989.

[14] J. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, 1996.

[15] N. H. E. Weste, K. Eshragian, *Principles of CMOS VLSI Design*, Addison-Wesley, December 2000.

[16] C. Edwards, *EDA Vendors Rethink Standard-Cell Libraries*, Electronics Times, June 2000.

[17] D. Pietromonaco, *Automating Cost-Effective Library Creation*, Integrated System Design, November 2000.

[18] http://www.research.compaq.com/wrl/projects/magic/

[19] http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/

[20] http://ece.www.colorado.edu/ ecen4228/spice/spice.htm

[21] http://james.cs.uga.edu

Appendix A

Hierarchy of the Fitness Function

The following table demonstrates the hierarchical nature of the fitness function (or cost function) which ensures that the GA fixes incorrect circuits before it optimizes them. The fitness values ranged from 0 to 5000 with 5000 representing the worst individuals. In practice, the higher bound is more than 5000 since an additional penalty is added at the very end in case the cell lies beyond the user specified cell limits. The GA tries to find a layout that produces the minimum fitness function. The penalty for "no influence" is discussed in Section 4.2.3. The penalty for "incorrect simulation" is proportional to the number of entries in the user specified truth table that were simulated incorrectly in SPICE. For working circuits the area of the cell is scaled so that it lies between 0 and 1. The minimum fitness function achieved for the inverter in Figure 4.9 was 0.666.

Table A.1: Hierarchical Fitness Function

Fitness Function returns	**For...**
5000	Wrong designs (shorted labels, overlapping transistors)
10 + penalty for "no influence" + penalty for DRC errors	Incomplete Designs (unconnected input, outputs or trans)
penalty for "incorrect simulation" + penalty for DRC errors	Complete Designs with DRC errors
Cell Area scaled by a constant factor	Correct Designs with no DRC errors

Appendix B

Look up Table for inter-layer distances

This table shows the values that were preset by the domain expert (Benjamin Bishop). These were used while calculating the distances between nodes in the influence check (Section 4.2.3) in addition to the manhattan distance between the nodes. MAGIC representations are 2 dimensional. In reality, the nodes sit on different layers in 3D space in the chip. Thus the inter-layer distances have to be also taken into account. The distances calculated by the 2D manhattan distance are made more representative of the true distances by adding the numbers from this look up table. Note that the inter-layer numbers between the same layer is 0[1].

Table B.1: Distance between CMOS Layers

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
(1) POLY	0	1	1	1	2	0	0	1	2	2	1
(2) NDIFF	3	0	3	1	2	4	4	2	2	0	1
(3) PDIFF	3	3	0	1	2	4	4	2	2	0	1
(4) M1	1	1	1	0	1	2	2	0	0	0	0
(5) M2	2	2	2	1	0	3	3	1	1	1	0
(6) NTRAN	0	0	2	2	3	0	1	1	1	3	2
(7) PTRAN	0	2	0	2	3	1	0	1	3	1	2
(8) PC	1	2	2	0	1	1	1	0	1	1	0
(9) NDC	2	0	2	0	1	3	3	1	0	1	0
(10) PDC	2	2	0	0	1	3	3	1	1	0	0
(11) M2C	1	1	1	0	0	2	2	0	0	0	0

[1]The top row has been labeled with numbers to reduce the width of the table. These numbers are labeled in Column 1.