Automated Performance Optimization of Custom Integrated Circuits

Thesis by

Stephen Mathias Trimberger

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1983

(Submitted January 6, 1983)

Acknowledgement

The completion of this work is due to my family and friends who endured the roller coaster ride and did not give up hope. I could not have survived the hard times had you not been there.

I would like to particularly acknowledge the one individual who, more than anyone else, has been responsible for my success throughout my life. He has been a competitor an instructor, a confidant, a model, an oracle, an assistant, a co-worker, a critic and a friend -- the perfect brother, Francis Michael Trimberger.

Many thanks to Flo Paroli, for providing perspective.

This work was funded by the Silicon Structures Project.

Abstract

The complexity of integrated circuits requires a hierarchical design methodology that allows the user to divide the problem into pieces, design each piece independently, and assemble the pieces into the complete system. The design hierarchy brings out *composition* problems, problems that are a property of the assembly as a whole, not of one single instance in the hierarchy.

Recent research has produced tools that automate part of the composition task -- the logical connection of the pieces. However, these tools do not ensure that signals driven over these connections will be driven sufficiently to give reasonable cycle speed of the resulting chips. It is easily possible to specify an assembly in which a small-sized gate is required to drive an enormous load. Parasitic capacitance of the wiring made automatically by the logical connection tool can be the dominant source of delay, so assembly tools can actually worsen the performance of the circuit and hide this fact from the designer.

When required to make large circuits, automated layout tools such as PLA generators can blindly make layouts that give abysmally poor performance. Here again, the delay is in a part of circuit that the designer did not specify, so it is hidden. Finding and correcting these problems is a difficult and time-consuming task in integrated circuit design, and one that consumes vastly more people's time and computer time than the simple assembly of the chip.

The task of guaranteeing that circuits meet performance specifications has been left mainly to the designer. Computer aided design has provided analysis tools, tools that tell the designer the performance statistics of the current design. It is then the designer's burden to interpret the performance statistics and use them as guides to make changes in the circuit.

This thesis views performance optimization as an electrical composition task. Poor performance as a result of mismatched loads on devices is a problem of composition and should be corrected by the composition tool. Such a tool is presented in this thesis -- a program that automatically sizes transistors in a symbolic description of a chip to match the load the transistors are driving. The results are encouraging: they show that delays can be cut by a factor of two in many current designs.

Table of Contents

Table of Figures

CHAPTER 1

Introduction

The complexity of integrated circuits forces a design methodology that encourages division of the problem into smaller pieces and subsequent assembly, or *composition* of the pieces into systems. This "divide and conquer" methodology is the basis of the design hierarchy, a hierarchy made up of instances of cells in which parent nodes in the tree contain the instances of the child nodes.

This division is made to facilitate the layout of the circuit and does not take into account the electrical properties. The result is that, although the design hierarchy gives a good abstraction for the layout of the chip, it is usually a very poor form for electrical optimizations such as minimizing delay and meeting current density limitations.

As work progresses in the synthesis of layout of integrated circuits, the optimization of electrical properties is falling farther and farther behind. Delay optimization is frequently done by hand, with designers making coarse estimates of loading by inspection and by assumptions about the circuits they are designing. Designers tend to ignore delay problems in most circuitry, making all devices minimum size, and tend to err on the side of conservatism for those gates which they believe will have to drive large loads. These oversized devices are wasteful of power and incur additional delay to drive them.

Typically load estimates are made by counting the number of gates on a

node, with no consideration of parasitic capacitance due to the wire. This was a reasonable assumption in the past, but the parasitic capacitances are beginning to dominate the gate capacitances in MOS, so designer's estimates are missing the mark. Symbolic layout, which is becoming increasingly popular, encourages connection by stretching which tends to hide the parasitic capacitances. These loads are not taken into account, so most of the circuitry on the chip runs more slowly than it should because the gates are minimum sized, and power is lost on those parts of the chip where the designer made a driver too large.

The problem with performance optimization is that the most important information, that having to do with the interconnection loading is not available to the designer until late in the design process. If a designer were to take this information into account, he would typically have to lay out the entire chip again. Since so much work is impractical and since the parasitic capacitance information is not readily available anyway, this whole problem is often ignored and slower, more power consumptive chips are the result.

Advanced chip assembly tools address physical design issues, but typically provide little assistance for these difficult assembly problems. These systems may make electrical optimizations more difficult, since they hide the implementation detail used to make the logical composition. Therefore, the designer cannot take into account the effects of the implementation when optimizing delays.

This thesis begins with a summary of the methods and tools currently used for performance optimization of integrated circuits. This summary leads to a discussion of delay models and an investigation of the tradeoffs between

delay and power consumption.

A program is presented for performance optimization that not only makes devices as large as they must be to drive their loads, but also saves power by altering gates off the critical path so they run slower and consume less power.

Examples of the use of the program show the value of automated performance optimization. For example, a performance improvement of a factor of two over hand designs has been achieved. Delay-power product can be improved by about twenty percent.

The later parts of the thesis describe in detail the algorithms used in the electrical optimizations and present alternatives and possible improvements to the algorithms. Finally, the role of performance optimization in a complete design system is discussed.

CHAPTER 2

Performance Optimization Issues

This chapter is a discussion of several issues surrounding performance optimization. It starts with a discussion of factors responsible for good performance in integrated circuits. Attention is focused on device modifications to improve performance, how it has been done in the past, and the rationale for the system described in this thesis.

Later sections deal with optimal delay in a chain of gates. A heuristic performance optimization method is presented and is compared to the optimal solution. Finally, the results are extended to accommodate full graph-like gate structure.

2.1. Where Does Speed Come From?

The performance of a circuit is affected by systems issues, circuit issues and implementation issues. Systems issues start with the way a chip fits into an overall system. The algorithms used to calculate the desired results are also systems issues, as are the geometric and electrical topology in the structures that implement the algorithms. This topology and the structure of the implementation is sometimes called the *floorplan* of the chip.

Circuit issues include the driving power and load on individual devices. Devices that drive large loads must be made large to accommodate the loads. Devices that do not drive large loads can be made small to save power.

Implementation issues center around the choice of implementation technology and the particular process parameters used for fabrication of the device in question.

Appropriate algorithms and efficient implementation structures are important to overall system performance. However, automating the choices involved is very difficult. The multiplicity of algorithms and implementation structures makes the choice of a good one still a design decision, often accompanied by high-level simulation. On the other end of the scale, process selection is important for good performance, but the means by which a process and the parameters for that process are selected is not one which can be easily automated. So we are left with circuit issues which are decided by relatively straightforward rules, but which are often ignored because the data are hard to collect.

2.2. Statement of the Problem

This work addresses circuit optimization issues in an nMOS technology. The program described in later chapters sets transistor and resistor sizes to balance the loads that those devices must drive, giving a faster circuit, within the constraints of the algorithms used for the function and the technology in which the function will be implemented. The designer may make changes in the structure and the algorithms in the circuit, and the circuit issues will be taken care of automatically.

2.3. Current Performance Optimization

Performance optimization has traditionally been the "weak sister" in integrated circuit design. Area and power optimizations are much easier to visualize and implement, and it has been a generally accepted belief that area optimization will give reasonable delay statistics. Delay optimization has only been addressed at a few industrial locations that specialize in high speed devices. Even there, more concern was directed to processing technology and algorithms than to circuit issues [Anderson 1982].

2.3.1. Semiconductor Industry Approach to Performance Optimization

In the semiconductor industry, circuit issues have been traditionally addressed in the design stage by coarse estimates of loading and by electrical simulation coupled with rules of thumb. Estimates and rules of thumb are usually stated in terms of gate counts and gate capacitances, without regard to parasitic capacitance. When parasitics are taken into account, it is typically done in a very rough manner since the length of interconnect is still unknown. Although these estimates may be improved later in the design cycle, the geometry of the transistors cannot be significantly changed.

Delay estimates are derived in three ways: gross estimates by the designer, simulation, and path delay analysis. Delay estimates for even moderately-sized chips are too difficult for a designer to carry out in his head, so automated estimators are gaining popularity. Electrical simulation can give good measures of delays [Daseking 1982], but it is expensive, so it is rarely carried out on large parts of a chip. Usually, the small cells of the design are

simulated electrically and it is hoped that the results of the cell simulations will not be invalidated by the composition. Unfortunately, electrical nodes cross the boundaries of those cells, and the purely cell-oriented simulation is inadequate to characterize the true performance of the cell.

Electrical simulation does not give the results the designer needs. Simulation tells the designer how good or bad the design is, it does not tell him how to correct a bad design. So the designer is caught in a very expensive loop of simulation and adjustment.

Path delay analysis [Bening 1982] is a relatively recent development in response to the inefficiency of simulation. Typical delay analysis tools find best case, worse case and normal delays for the paths through the chip, directing the designer's attention to problem areas. Path delay analysis systems are considerably faster than simulation and they provide information in the form of delays, which are a more reasonable starting point from which to optimize the circuit. However, path analysis systems merely point out problem areas, they do not help correct them. After these problems are corrected, the entire process must be repeated. These design iterations are expensive and time consuming.

An interesting exception to the standard industry performance optimization approach is a synthesis tool described in [Agule 1977] and [Ruehli 1977]. The designer creates a gate array design and imposes some timing constraints on the chip. The system adjusts the assignment of gates in a gate array system so gates that drive small loads are assigned to less powerful gates in the gate array, saving power. This system is similar to the one described in later chapters of this thesis, but is constrained to work with a gate array

implementation.

2.3.2. University Approach to Performance Optimization

The relatively recent explosion in the university involvement in integrated circuit design has brought few new ideas on performance issues. With few exceptions, the only work in the university community in improved performance has concentrated on algorithms and structures for high-speed or parallel processing.

The most notable contribution of the university community to improving performance is the Spice simulator [Cohen 1978]. Some work on logic design systems has addressed performance issues [McWilliams 1978], which may be applicable to integrated circuit design. More recently, [Penfield 1981] derived some estimates for distributed resistance and capacitance effects in integrated circuits. These estimates have application to simulation and delay path analysis.

However, the university community in general has accepted simplified delay models for integrated circuit design and has produced few tools to aid circuit delay optimization.

One powerful design concept from the university community is language-based integrated circuit design. The use of a programming language allows cells to be *parametrized:* a cell can be referenced with parameters to set cell size and drive power. If the cells are parametrized properly, a system can be built which determines signal loading and calls cells with the parameters necessary for optimal drive, thus minimizing delay. Although such possibilities have been recognized since the first embedded language system

was proposed [Locanthi 1978], no such systems optimize delay, although Bristle Blocks [Johannsen 1981] parametrizes power and ground bus widths to avoid current density problems.

This light treatment of performance issues may be attributed to the university community's concentration on fast turnaround of parts and the subsequent acceptance of second-rate in all integrated circuit design parameters. There seems to be little effort to gain optimal area, power dissipation, or speed of operation. For the sake of expediency, most university designers use simplified geometrical design rules from [Mead 1980] and skip precise timing analysis and optimization.

Unfortunately, although the simplified geometrical design rules yield designs that are in the best cases within about twenty percent of the most dense layouts, simplified delay models often yield designs that are a factor of two slower than optimum. The simplified design rules may be acceptable but the simple delay optimization appears to be too simple. More accurate delay optimization could improve the resulting designs considerably, but the cost of traditional industrial solutions to performance problems are not acceptable to university designers. These solutions require too much computer time and too much delay to fabrication.

2.3.3. The Inadequacies of Current Performance Optimization Practices

The industrial approach of massive simulation and evaluation is too expensive and only leads designers to problem areas. Solutions to the problems require hand modification of the design requiring all checking to be re-done.

universities give up high performance for the sake of fast turnaround. Fast turnaround requires automating layout or interconnection or both. Programmable Logic Array (PLA) generators produce slow-operating PLAs, and automated interconnect systems can hide the interconnection delays from the designer. Correcting these problems takes time, so they are not treated in the detail they require. This practice costs a factor of two or more on many designs, and cost an estimated factor of ten on a recent project [Foderaro 1982].

2.4. A New Way to Address Performance Optimization

This thesis explores the circuit performance optimization question in a system that automatically sets device sizes depending on the load which those devices must drive. Such a system has many advantages over existing and non-existent delay optimization methods. This section introduces the concepts and the goals of the system.

2.4.1. Automated Sizing of Transistors

A system that automatically sets device sizes does not seriously impede the fast turn-around desired by the university community as long as the program runs in an insignificant amount of time compared to the fabrication delay. A fully automated system can meet this restriction rather easily. Such a system also performs the task the designer wants, actually optimizing the delays instead of telling the designer what delays there are in the circuit and requiring him to make the changes. This system could be used as a code optimizer for a "silicon compiler".

It has been suggested that symbolic layout could be used as an interchange form so chip area could be optimized separately for each process line on which the chip is to be fabricated. This optimization could be done for delays also, using the ideas presented in this thesis. Since each process line has its own process resistance and capacitance parameters, the constants in the program can be changed for each new process line. This optimization could have dramatic effects on the performance of second-sourced parts, and for parts after a change in the fabrication process.

2.4.2. Changes in Device Sizes Mandate Physical Changes

The optimizer makes geometrical changes in the circuit as well as electrical changes. Because device sizes change, the geometry of the circuit changes. This is disastrous in systems that are based on hard mask geometry, as most are. The answer, of course, is to make the changes in a symbolic form. Since the symbolic components can be moved, the circuit must be re-run through the symbolic area optimizer after the delay optimization has been done. Automated changes in device sizes require automated changes in the physical location of components.

Without the symbolic form, changes in device sizes require a designer to alter the design by hand. Such changes then require that the design be checked for correctness. And, of course, the simulation of performance estimation must be run again to check the changes. This loop is time consuming and expensive.

2.4.3. Targeted Delay Optimization

It is unreasonable to attempt to develop a circuit at the absolute minimum delay. Such a circuit would consume an enormous amount of power and would require an unreasonable amount of chip area. An algorithm that attempted to make absolutely fast circuits would make unusable circuits. The algorithm described in this thesis gains a reasonable amount of delay optimization without seriously affecting the power or area statistics.

Delay optimization need only be done on the delay critical path. Other paths can be made as slow as desired as long as they do not create delays longer than the critical path delay. This saves power on parts of the circuit that are off the critical path.

2.5. Delay Models

This section examines several delay models and discusses their usefulness in a system that optimizes delays in circuits. Nearly all of the work on delay models has been for simulators and delay path analysis systems, but much of it is applicable to automated performance optimization.

2.5.1. Transistors

A transistor delay can measured as the delay from the time when the voltage on the input of a gate reaching a value to the time when the output reaches that value. A common voltage at which these delays are measured is the voltage at which the voltage on input of an inverter equals the voltage on the output, V_{EQ} in figure 2.1. [Pilling 1972a] [Nham 1980]. Others have used

other points, such as the half-way point between the high and low voltage [Putatunda 1982].

Given that delays are to be measured from some such point, there are two more problems which must be addressed to estimate delays: the actual equations involved to find the times at which the voltages cross that point, and the measures of the resistances and capacitances, which are the input to those equations.

2.5.2. Delay

Starting from a standard measuring point, it is possible to translate simulated voltages into delays. There is a wide spectrum of choices for this

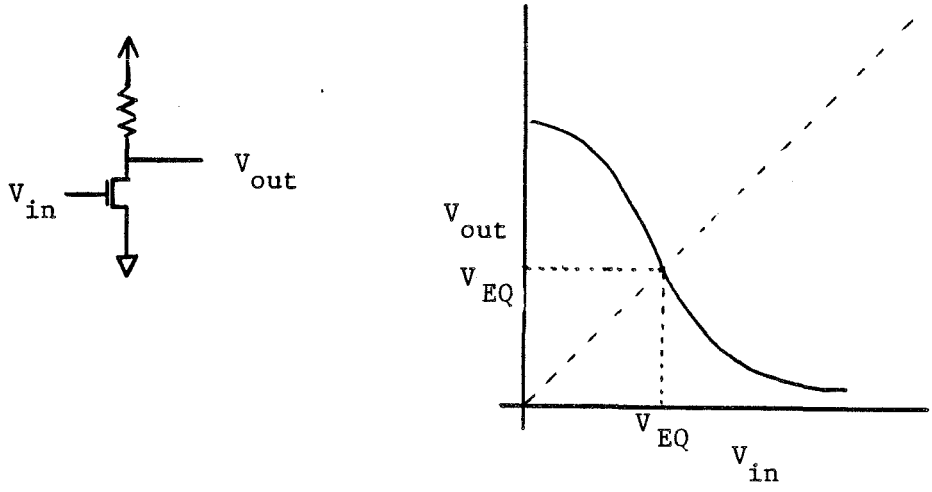


Figure 2.1. Input Voltage Equals the Output Voltage.

translation, ranging from very difficult, precise calculations to very simple estimates. A rather good estimate of the voltage, V, at some time, t, can be gotten from the equation:

$$V(t) = \int_{-\infty}^{t} \frac{1}{R(\tau)C(\tau)} V_i(\tau) e^{-(t-\tau)/R(\tau)C(\tau)} d\tau$$

where R is the resistance discharging the capacitance, C, and V_i is the input voltage which may vary over time. For digital MOS circuits, the resistance in the equation is the resistance of a transistor, which varies with the voltage on its gate, hence the dependence on τ. The capacitance on the node is also dependent on the voltage on the node, so it also varies over time. There is even more complication: there may be more than one driver on the node, as is the case in NOR structures in circuits, so the voltage may be the sum of many of such equations.

Furthermore, the device that drives the node high is different than the device that drives it low. The difference in delay is significant in nMOS circuits and has led some researchers to describe the node delay with two separate equations, one for rise time and one for fall time [Koppel 1978]. This practice yields a combinatoric explosion in the number of different delays for an output node, since the delay at the output is a function of the values on the inputs of the circuit. Therefore, this distinction between rise and fall times is not always made.

The delay model used in this thesis takes the rise time for all nodes at all gates, because the rise time is longer than the fall time. The resulting delay estimate is rather pessimistic, since half of the inverters in a chain would be falling instead. It gives $nk\tau$ for the delay of a chain of inverters, where n is

the number of inverters in the chain, k is the ratio of the pulldown resistance to the pullup resistance in an inverter, and τ is the transit time through a transistor, as described in [Mead 1980]. The true longer delay (most rising signals) is $\left\lceil \frac{n}{2} \right\rceil (k+1)\tau$ for n even, and $\left\lceil \frac{n-1}{2} \right\rceil (k+1)\tau + k\tau$ for n odd.

The estimate, $nk\tau$, is more accurate for small chains of gates (n small) and remains less than twice the true value in the limit for long chains. In the system described later, this estimate is used as a comparison with other, similarly pessimistic delays. Therefore, the pessimism of this simple model is not catastrophic as long as there is no great mismatch in the number of gates in the chains whose delays are being compared. In addition, since the estimate is more pessimistic for longer chains, the system will tend to select a longer chain of gates as the critical path over a shorter one -- a situation that is not nearly as bad as the converse. This single "figure of merit" simplifies comparisons immensely.

There is at least one of the delay equations, above, for each node in the circuit. Since all must be solved simultaneously, we will be well served to find some simplifications.

The integral looks rather forbidding, but the real problem with this calculation is the need to know the input voltage, resistance and capacitance over for all time. These are dependent on other nodes in the circuit, so each node is dependent on many nodes, all of which are described by similar integrals.

First, if we assume that the power supply does not vary, we can make V_i constant. Assuming that the resistance of the pullup transistor is constant

and that the capacitance on the node does not vary significantly with voltage, we get the equations below:

$$V(t) = \frac{V_i}{RC}\int_{-\infty}^{t} e^{-(t-\tau)/RC}\,d\tau = V_i\left(1-e^{-t/RC}\right)$$

We can solve for t when the $V(t)$ is the voltage at which the output of a standard inverter is equal to its input, V_{EQ}:

$$t = -RC\ln\left(1-\frac{V_{EQ}}{V_i}\right) = k_{EQ}RC$$

So the delay can be expressed in terms of the RC delay constant of the node. Notice that the delay is proportional to the RC time constant regardless of the special voltage measured. If we measure from the midpoint voltage, we would simply have a different constant. This simple result is valid when a perfect resistance is discharging a perfect capacitance over a perfect conductor. Transistors do not have perfect resistance and perfect capacitance, so some ad-hoc approximations with higher-order terms have been proposed, for example [Koppel 1978]. Since perfect conductors are perfectly rare in integrated circuits, I now address wire models.

2.5.3. Wires

A wire is not a pure conductor, but has some resistance and capacitance. These are frequently called *parasitic*, a term that carries a connotation of secondary importance. However, parasitic capacitance is beginning to dominate the gate capacitance in integrated circuits, and a delay model that does not take it into account will be inaccurate.

To be absolutely precise, the voltage along a conductor should take into account the distributed nature of the parasitics. A voltage diffuses down a wire with a behavior described by a differential equation:

$$RC\frac{\partial V}{\partial t} = \frac{\partial^2 V}{\partial x^2}$$

It is unreasonable to solve this diffusion equation for every wire in the design. Until recently, wires were often treated as prefect conductors, and in some integrated circuit technologies, this is still a valid assumption. But MOS technologies require a more accurate treatment of wires. Therefore, estimates of different precision have been used for wires. Some of these models are shown in figure 2.2.

A pessimistic estimate is to put the entire wire capacitance after the wire

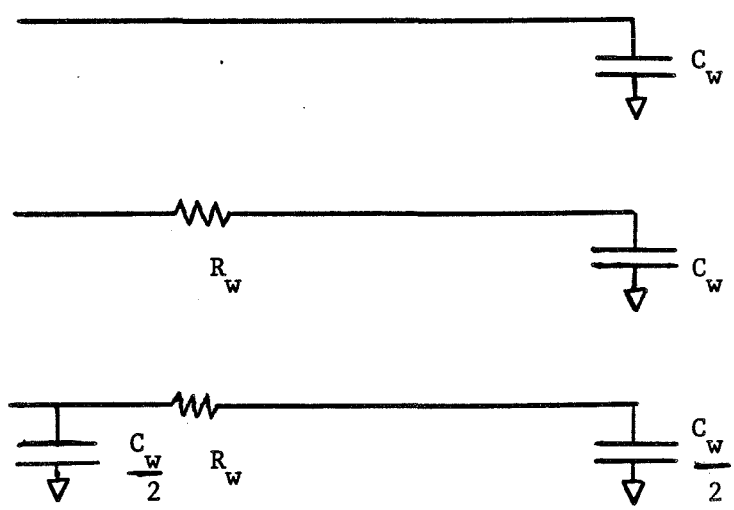


Figure 2.2. Wire Models

resistance, an optimistic estimate is to put the capacitance before the resistance or eliminate the resistance altogether. A compromise is to divide the capacitance, putting half in front and half after the resistance. These empirical models make reasonable approximations to the diffusion equations, above, for most situations.

To choose an acceptable simplification, let us examine electrical parameters for MOS wiring and transistors, figure 2.3. The capacitances of wiring layers in MOS are typically one less order of magnitude than gate capacitances. Therefore, any wire model must take into account the wire capacitance.

On the other hand, wire resistances are all several orders of magnitude less than the transistor resistance, so for most circuits, wire resistance effects are slight. The resistance of a polysilicon wire, the most resistive wiring layer in nMOS, is typically less than one one-hundredth the gate resistance, so differences in the models are significant for polysilicon wire lengths greater than about few hundred lambda. Longer diffusion and much longer metal wires are needed before their wire resistances are significant.

Simulations indicate that short wires in current technologies can accurately be modelled with a purely capacitive model. Longer wires cannot be

Transistor Capacitance	4.0×10^{-4} pf/μm^2.
Diffusion Capacitance	1.0×10^{-4} pf/μm^2.
Polysilicon Capacitance	0.4×10^{-4} pf/μm^2.
Metal Capacitance	0.3×10^{-4} pf/μm^2.
Metal Resistance	0.03 Ω/■.
Diffusion Resistance	10 Ω/■.
Polysilicon Resistance	15-100 Ω/■.
Transistor Resistance	1.0×10^4 Ω/■.

Figure 2.3. MOS Electrical Parameters from [Mead 1980]

adequately modelled so simply, however, and the simplified model is not applicable to these wires.

Wire delays do not scale well. The resistance per square and the capacitance per unit area both increase in proportion to the scaling. The size of the allowed chip area is unchanged, so wires may be just as long. The result is that the resistance per unit length of a wire increases as the square of the scaling factor, due to thinner and narrower wires. The capacitance per unit length remains constant. The transit time of a transistor decreases by the scale factor. So the wire delay increases as the cube of the scaling factor relative to the device transit time. Wire delays are rapidly getting more important than gate delays. In current technology, wire delays are a problem in high-performance circuits, and must be considered throughout the design process. Future technologies will require consideration of wire resistance for long resistive wires.

Currently, in circuits where absolute high performance is not essential, the wire delays are not considered during the design of the chip, but the resistances are calculated after the design is complete to determine whether or not the wire delay is significant. Wire delay is only considered on very long polysilicon and diffusion wires, and blatant conservatism guides the designer in addressing the wire resistance problems. This conservatism is prompted by the disastrous effect of ignoring wire resistance in very long wires, since the delay increases as the square of the distance.

An investigation by [Bilardi 1981] of the necessary complexity of wire models concludes that the capacitive model is adequate for current and future MOS technologies. This work ignores the scaling of the thickness of the wires,

thereby diminishing the effects of long wires on delay. Although this implies that the results are not conclusive, it is safe to say that although the diffusive effect of long wires will limit the speed of operation for integrated circuits in the future, it is not now a dominant problem.

Although the wire resistances can be safely dismissed for the near future, the resistance on a wire as a result of a pass transistor cannot. To cope with this problem, [Penfield 1981] produced some bounding equations on the speed of signal propagation along a wire and within a RC tree network. These equations can be used to obtain bounds on the signal delay on a wire, and they have been used to get an estimate of the path delay [Putatunda 1982]. This method may be less accurate than the simple models described above, since it uses an average of *bounds* on delay, however, it is more accurate over a large range of wire characteristics because it can take into account distributed resistances and capacitances. This more complex model is also more difficult to compute. The difficult task is to choose a model that is sufficiently accurate and sufficiently simple for use in a tool.

2.5.4. A Simple Solution

The delay model for optimization serves two purposes. First, it is used to estimate delays in the circuit to direct the optimization task, similar to the delay path optimization done at many industry locations. Second, it is used in reverse to reduce the delay in part of the circuit. It is desirable to have one delay model for both functions. In addition, in addressing the conceptual basis for performance optimization, we wish to use a simple delay model.

Nearly all of the delay models were proposed to address simulation and delay

estimation issues. In these applications, good quantitative models are needed, and most of the computational complexity of those systems has been included to get more accurate timing by taking into account parameters that might be of no importance in the task of actually optimizing the performance of the circuit. Also, electrical simulators are targeted to analog circuits as well as digital, making the accuracy of the delays that much more serious. In a delay optimization system, there is no need for the delay model to be a good quantitative model, only a good qualitative model. Although we attempt to find a minimum delay, it is not necessary to know what that minimum is. In addition, this delay optimization system is targeted to digital systems, so complexity added for analog circuits is not relevant.

Simpler models can be used to simplify the programming and speed up the execution without undue reduction in the accuracy of the results. A tool with a fast turnaround can allow the designer to experiment with different structures and find a good one. Therefore, a very simple performance model is used in the performance optimization which is discussed in the remander of this thesis.

Delays are assumed to be measured from a standard voltage point. That point may be V_{EQ}, but the precise value is not important. Therefore the delay of a gate is simply a constant times the product of the resistance of the pullup resistor on the gate and the capacitance of the output node of the gate:

$$D = kRC_L$$

Since the resistance of the pullup transistor in a gate is proportional to the

resistance of the pulldown transistor when it is turned on, the rise time and fall time are related by a constant factor, so there is no need to distinguish between the two. In this simple model, a wire is modelled as a lumped capacitance which is added to the capacitance of the gates on the node. The wire resistance is ignored because it is insignificant in most cases, as discussed in the "Wires" section, above.

$$C_L = C_{gates} + C_{wires}$$

The delay of a chain of gates is the sum of the delays of the individual gates.

$$D_{chain} = \sum_{i=1}^{N} D_{gate_i}$$

The lumped capacitance, zero resistance model of wires allows us to treat wire capacitance and gate capacitance uniformly, simplifying the algorithms. A reasonable improvement in the system would be to model wire delays and their associated loads on the gates. This improvement is discussed in a later chapter.

The simplification gives a reasonable qualitative estimate of the delays. It is rather optimistic when dealing with long wires, on which the delays are proportional to the square of the length, because it does not take into account the wire resistance. Therefore the performance optimizer will underestimate the delay in a long wire, possibly missing that path as a critical path, possibly driving it insufficiently. But since the delay of the path is in the wire, that path will not go faster if it is driven harder.

In current integrated circuit design, wire resistance effects are significant only in very long polysilicon wires. Long polysilicon wires are used in local

clock routing because no layer change is needed to make a pass transistor. They are also used in PLAs, so no layer change is needed to make a gate. Also, long polysilicon wires may occur in the wires produced by channel routers, which make connections in metal and polysilicon, avoiding diffusion because of its large capacitance. This simple model will underestimate the delay of signal propagation in these cases, but the error will not be very large.

A more serious limitation occurs because pass transistor resistance is ignored. Delays due to pass transistor resistance are more serious, and may lead to serious underestimations of delays.

As integrated circuit geometries get smaller, the relative length of wires can be expected to become larger, so the resistive effects will eventually become more serious. Therefore, wire resistance must be taken into account in systems in the future. This and other improvements are discussed in the chapter titled "Performance Optimization Options".

The inaccuracy of the simple wire model does not affect the algorithm used to optimize the delays, but does affect the resulting device sizes. Inclusion of a more accurate wire model is akin to inclusion in a Sticks system of more accurate design rules. The result is a better-optimized circuit, the cost is more computation to find the optimization.

2.6. Delays in a Chain of Gates

In this section, I derive new equations to find the sizes of gates in a chain which will minimize the delay of a signal through the chain, a combination of

delay and power consumption, and the delay-power product.

Following that, I present the fundamental algorithm used in Andy for sizing gates. This algorithm does not yield optimal delay nor does it give optimal delay-power product. However, it does yield reasonable results, as will be shown from comparison with the results from the equations. Later sections deal with the comparison of the heuristic to the optimum and with more complex gate and wire structures.

2.6.1. Optimum Delay

We are given a chain of gates, shown in figure 2.4. The chain is driven by a pulldown transistor with a width of w_0, and the chain must drive a load capacitance of C_L. The task is to set the sizes of the transistors in the gates in the chain to minimize signal delay across the chain.

[Mead 1980] states that for minimum delay, the number of gates in a chain driving a large load should be chosen so that each one is larger than the previous one by e, the base of the natural logarithms. The minimum is broad and flat around e, and larger fanout yields area and power saving, so in practice, this number is usually between four and eight.

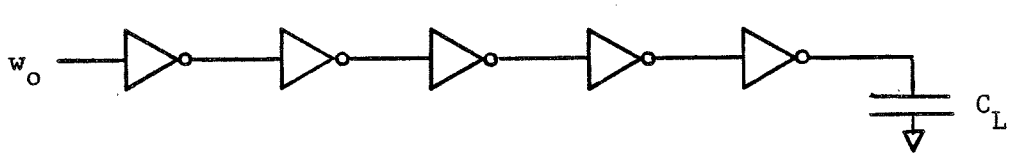


Figure 2.4. A Chain of Gates Between a Driver and a Load

However, we are faced with a slightly different problem: the number of gates in the chain is given and we must find the device sizes that minimize delay. It seems obvious that each should be larger than the previous one by a constant, and this is exactly what the equations, below, will show. But we will also find from the equations how to choose gate sizes to find an optimum of a function of power and delay and for the minimizing the delay-power product. Gate sizes in these cases do not vary by a simple constant factor.

2.6.2. Equations for a Single Gate

An inverter gate like the ones shown in figure 2.4 has two driving transistors, a pullup transistor and a pulldown transistor. In nMOS, the sizes of the transistors is related by the following equation:

$$\frac{l_{pu}}{w_{pu}} = k_{pu}\left(\frac{l_{pd}}{w_{pd}}\right)$$

where w and l are the widths and lengths of the transistors and the subscripts pu represent the pullup and pd the pulldown. k_{pu} is the ratio of the inverter pullup width to length ratio and the inverter pulldown width to length ratio for nMOS, which is usually taken as four. The length/width ratio is proportional to the resistance of the transistor, so we can express the resistance of the gate as a constant times resistance of the pulldown transistor:

$$R_g = k_r'\left(\frac{l_{pu}}{w_{pu}}\right) = k_r\frac{l_{pd}}{w_{pd}}$$

The capacitance with which we are concerned is the capacitance of the pulldown, which can be expressed as:

$$C_g = k_c l_{pd} w_{pd}$$

Where k_c is the capacitance per unit area of a transistor. Notice that we have the resistance and the capacitance in terms of the dimensions of the pulldown transistor.

2.6.3. Equations for a Chain of Gates

Let w_i and l_i be the length and width of the pulldown transistor in the i^{th} gate in a chain. Write R_i for the resistance of the i^{th} gate and C_i^{out} for the capacitance the i^{th} gate must drive, and C_i^{in} for the capacitance of the gate itself. The power dissipated by a gate can be measured by the resistance of the transistors in the gate:

$$P_i = k\left(\frac{1}{R_i}\right) = k_p\left(\frac{w_i}{l_i}\right)$$

where k_p is the constant that converts the gate w/l ratio to a measure of power consumption. It absorbs the constant in the gate resistance equation. As discussed earlier in this chapter, the delay of a gate can be approximated as a constant times the resistance of the driving transistor times the output capacitance:

$$D_i = kR_i C_i^{out} = k_d\left(\frac{l_i}{w_i}\right) C_i^{out}$$

where k_d is the delay constant. The capacitance of the i^{th} gate is proportional to the area of the pulldown transistor:

$$C_i^{in} = k_c l_i w_i$$

where k_c is the constant for capacitance. It should also be noted here that

we are assuming perfect conductors for clarity, so the output capacitance of the i^{th} stage is the gate capacitance of the $i+1^{st}$ stage:

$$C_i^{out} = C_{i+1}^{in}$$

We can assume that the lengths of the transistors in the pulldowns of the gates are set to some minimum length, l_0. Therefore, we need deal with widths of transistors only. The power, delay and capacitance of the i^{th} gate become:

$$P_i = k_p\left(\frac{w_i}{l_0}\right) = k'_p w_i$$

$$D_i = k_d\left(\frac{l_i}{w_i}\right)C_i^{out} = k'_d\left(\frac{1}{w_i}\right)C_i^{out}$$

$$C_i^{in} = k_c l_0 w_i = k'_c w_i$$

C_L can be rephrased now in terms of a transistor width that would have that capacitance:

$$C_L = k'_c w_{N+1}$$

2.6.4. Equations for Total Delay and Power

We can now form the equations we wish to optimize. The total delay of the chain is the sum of the gate delays:

$$D_{TOT} = k'_d \sum_{i=0}^{N} \frac{C_i^{out}}{w_i} = k'_d \sum_{i=0}^{N} \frac{C_{i+1}^{in}}{w_i} = k'_d \sum_{i=0}^{N} k'_c \frac{w_{i+1}}{w_i} = K \sum_{i=0}^{N} \frac{w_{i+1}}{w_i}$$

where K is the product of the constants, $k'_d k'_c$. The equation for the total

power of the circuit is the sum of the power consumed by each of the gates:

$$P_{TOT} = k'_p \sum_{i=1}^{N} w_i$$

2.6.5. Solving for Minimum Delay

The widths for minimum delay can be found by differentiating the total delay:

$$\frac{\partial D_{TOT}}{\partial w_i} = \frac{\partial \left[K \sum_{i=0}^{N} \frac{w_{i+1}}{w_i} \right]}{\partial w_i} = K \left[\frac{1}{w_{i-1}} - \frac{w_{i+1}}{w_i^2} \right]$$

Setting that equal to zero, we find:

$$\frac{w_{i+1}}{w_i} = \frac{w_i}{w_{i-1}}$$

This states that the ratio of any adjacent pair of gates must be identical to the ratio of any other adjacent pair of gates for the minimum delay solution. This derivation verifies the assumption made in [Mead 1980].

This solution can also be used to find the optimal gate sizes, given the initial conditions w_0 and $w_{N+1} = C_L / k'_c$, and given a fixed N, the number of stages in the chain. Since each gate must be larger than the previous one by a constant, the width of the i^{th} gate can be expressed as:

$$w_i = w_0 \left[\frac{w_{N+1}}{w_0} \right]^{\frac{i}{N+1}}$$

So the total delay for the chain of gates is:

$$D_{TOT} = K\,(N+1)\left(\frac{w_{N+1}}{w_0}\right)^{\frac{1}{N+1}}$$

2.6.6. Solving for a General Delay and Power Function

Usually, designers are concerned with power consumption as well as speed of a circuit. We can take power consumption into account by minimizing a function of power and delay:

$$J = P_{tot} + \alpha D_{tot}$$

Here, α is a parameter that weights power consumption. Large α indicates great concern with power, α=0 is the case discussed above where power was of no concern. Differentiating this function, we get:

$$\frac{\partial J}{\partial w_i} = K\left(\frac{1}{w_{i-1}} - \frac{w_{i+1}}{w_i^2}\right) + \alpha k'_p$$

Setting this equal to zero and solving, we find the relationship for the sizes of transistors in this case:

$$\frac{w_{i+1}}{w_i} = \frac{w_i}{w_{i-1}}\left(1 + \frac{\alpha k'_p}{K} w_{i-1}\right)$$

Each stage of the chain is larger than the previous stage by the constant factor plus an additional term that depends on the size of the previous gate. Thus to drive the same load from the same number of gates, the scale factor is smaller, there are fewer large, power consumptive gates and more small low-power gates.

This recurrence is much more difficult to solve than the one for minimum delay. Figure 2.5 shows plots of numerical solutions of transistor size versus

stage in the array for a fixed load and for α = 0, 0.1, 0.3, 0.7 and 1.0. α = 0 is the minimum delay solution.

The graph in figure 2.5 plots the stage of the gate in the chain of gates versus the log of the width of the gate, so the minimum delay solution appears as a straight line. The delay of the chain is related to the length of the path on the graph:

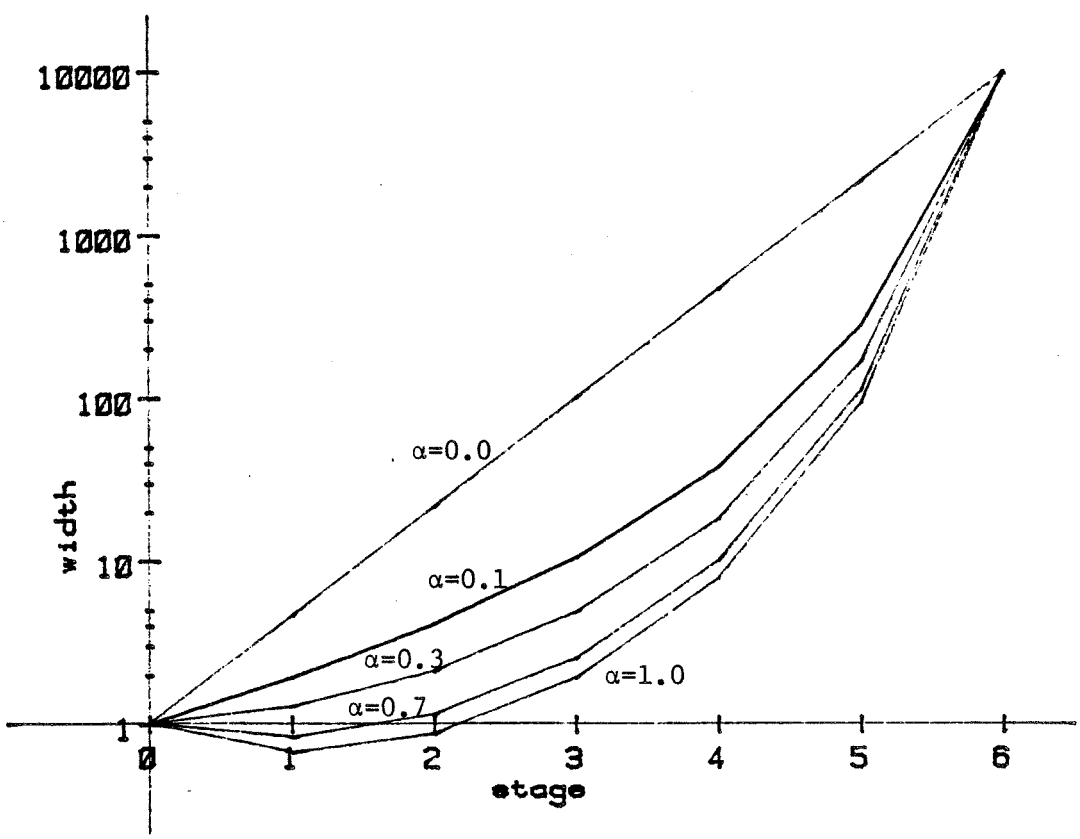


Figure 2.5. Plots of Log of Transistor Width Versus Stage for Different Values of Alpha.

$$D_{TOT} = K\sum_{i=0}^{N}\frac{w_{i+1}}{w_i} = K\sum_{i=0}^{N}e^{\ln(w_{i+1})-\ln(w_i)} = K\sum_{i=0}^{N}e^{m_i}$$

where m_i is the slope of the line in figure 2.5 between the i^{th} and the $i+1^{st}$ stage. So, for $m_i>0$, longer paths imply greater delay.

The power is related to the area under the curve by:

$$P_{TOT} = k'_p\sum_{i=1}^{N}w_i = k'_p\sum_{i=1}^{N}e^{y_i}$$

$$A = \sum_{i=0}^{N}\frac{y_i+y_{i+1}}{2} = \frac{y_0}{2}+\sum_{i=1}^{N}y_i+\frac{y_{N+1}}{2}$$

where y_i is the y-value on the plot, $\ln(w_i)$. This means that higher curves imply greater power consumption.

2.6.7. Solving for Delay-Power Product

A value used frequently to measure the quality of a design is the product of the delay of the circuit times the power it dissipates: the delay-power product. We can find the optimum in this case:

$$\frac{\partial(P_{TOT}D_{TOT})}{\partial w_i} = P_{TOT}\left[K\left[\frac{1}{w_{i-1}}-\frac{w_{i+1}}{w_i^{\,2}}\right]\right] + D_{TOT}(k'_p)$$

which resolves to:

$$\frac{w_{i+1}}{w_i} = \frac{w_i}{w_{i-1}}\left[1+\frac{D_{TOT}k'_p}{P_{TOT}K}w_{i-1}\right]$$

This equation is the same form as the one above with $\alpha=\frac{D_{TOT}}{P_{TOT}}$. Note, however, that this equation is deceptively simple, since D_{TOT} and P_{TOT} both depend on the widths of the gates.

2.7. A Simple Algorithm for Nearly Optimizing Delay in a Chain of Gates

In order to drive a large capacitive load with minimum delay, a designer inserts a string of ever-larger buffers. The number of buffers and the ratio of their sizes is determined by the optimal *fanout* for the design constraints. For minimum delay, each stage should be larger then the previous stage by a constant *fanout factor*, as has just been shown.

If the circuit has already been specified, the number of restoring logic gates is fixed, and *optimal* is now in reference to the given number of stages in the circuit. The equations above still hold, and the minimum-delay solution is the one in which each stage is larger than the previous one by a constant factor, as shown in figure 2.6. If the number of stages is not optimal, the fanout factor will be either larger or smaller than the optimal fanout factor, but the circuit will still perform with minimum delay for the number of stages.

The optimal delay and power equations require that all gates in the chain be sized simultaneously, since the number of gates in the chain will not necessarily have any relation to the optimum number. The fanout factor will

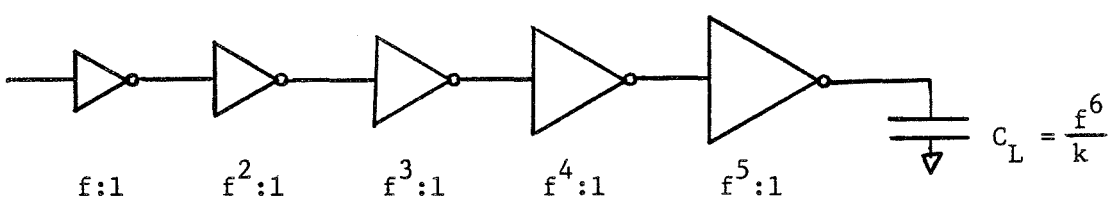


Figure 2.6. Optimal Fanout for a Chain of Gates.

have to be computed from the number of gates in the chain. This computation is more difficult with more complicated nets of gates, making the calculations very time consuming indeed.

A simpler heuristic algorithm is presented here which is not guaranteed to give an optimum. This algorithm is used in the system described in subsequent chapters, and a detailed description of the implementation of the algorithm is given there.

2.7.1. Heuristic Delay Optimization

If we work backward through a circuit, we can look only at the load on a node and at the gate driving it to size the gate. The gate size is set to the optimal *fanout factor* value, which will only give the optimum delay if there are exactly the right number of gates in the chain. The rest of the chain is ignored, and the algorithm works backward, re-sizing the nodes that drive the inputs to the gate. No transistor can be set to less than a minimum size, so with a long chain in which the optimal result would have a fanout factor less than the optimal (i.e. "too many" gates in the chain for optimal fanout), some gates in the chain will be minimum size and the later gates will increase in size at the optimal fanout rate, as can be seen in figure 2.7.

This solution is lower speed than the optimal. However it can be seen when comparing the graph in figure F to the graph in figure 2.5 that this solution is also lower power than the original. The relationship between the delay and power consumption by this chain of gates and the delay and power consumption of a chain of gates with a particular α depends on the number of gates in the chain.

It also bears notice that this ramped driver is closer to what is done by human designers. Most of the gates are minimum size, and only those near the large load are made larger to accommodate the load. This yields a lower power solution, as has been noted, but that is secondary. More importantly, this yields a smaller area solution and a more regular solution, since gates in the interior of the chip may be part of a large regular array (for example a memory array). Modification of those arrays would significantly increase the complexity of the design and complicate the logical assembly of the chip.

When the number of gates in the chain is less than the optimal number, we reach a solution where the first gate in the chain may be very large

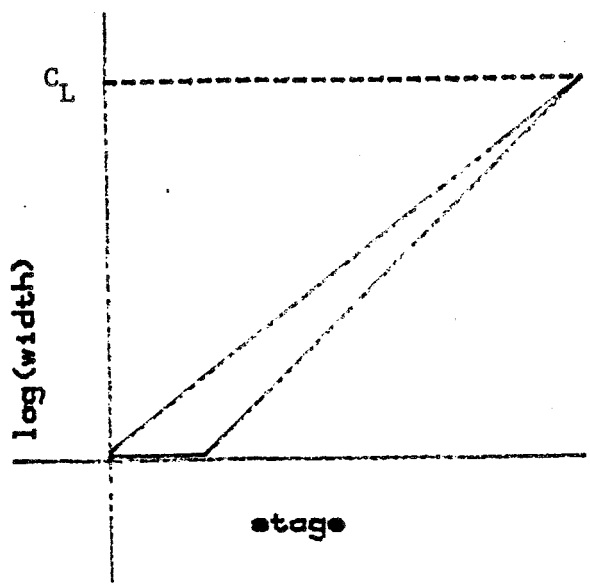


Figure 2.7. Gate Size Versus Stage for Number of Gates Greater Than Optimum for Fanout

compared to its driver (see figure 2.8). This yields a solution that is not only slower but more power consumptive than the optimal delay solution.

We can address the problem by making the assumption that the chain really does not begin there, because we are sizing gates for the whole chip. The gate that drives this first gate will also be sized in this manner, so it will be put along the optimum fanout line, as shown by the dotted line in figure 2.8.

Of course, there must be a first gate *somewhere*, but on the chip, the first gate takes its input from one of the input pads. We can assume that the driving power of an input pad is very large indeed, so there will be few, if any, cases where this happens. However, this case does happen when an attempt is made to drive off-chip much more strongly than the signal that comes on chip, as is done with line drivers. Since line drivers constitute such a small part of the LSI design problem, it seems safe to relegate them to the "special case" category. Thus this algorithm does not work properly on a line driver where there are not enough gate stages to drive the output with the optimal fanout factor between stages.

There is a sub-case of the case above in which there were too many gates in the chain between the load and the driver. Figure 2.8 had the first driver at minimum size. If the driver were larger than minimum size, an input pad for example, the output of the algorithm would be as seen in figure 2.9. The straight line shown by optimum delay might be desirable. More desirable might be the curve in which gates are made smaller until they are the same size as the input driver. However, this algorithm produces the lowest curve, corresponding to lowest power solution. The shape of this curve is similar to the shape of the curve in figure 2.5 above with large α. One interpretation of

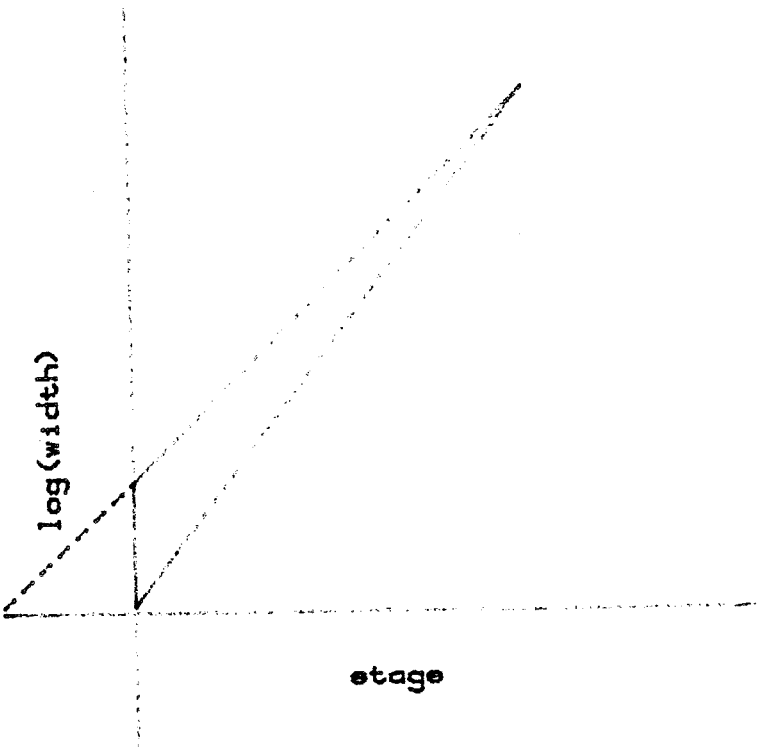


Figure 2.8. Gate Size Versus Stage for Number of Gates Less Than Optimum for Fanout.

this curve is that the original driver is too large. It can also be interpreted as great concern with power. Whether or not this represents "over concern" with power is dependent on the application, but it should be noted that this sharp drop from the large driving power of the input pad to a minimum-sized transistor is frequently done in chip designs made by human designers. It may not be good, but it is typical.

2.7.2. Power Optimization

Once all gates have been set to optimize delay, the path through the circuit with the largest delay must be the critical path. Gates off the critical path

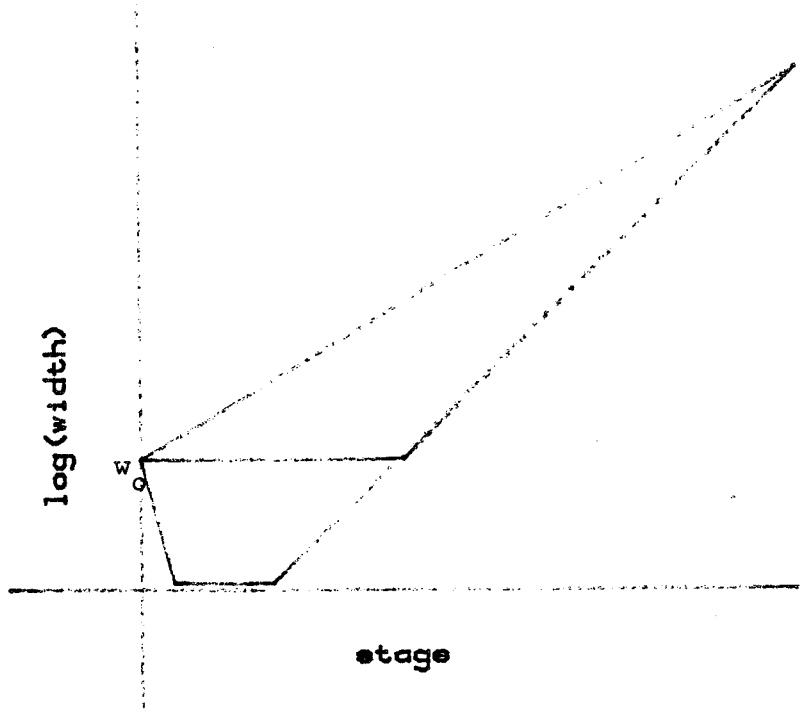


Figure 2.9. Gate Size Versus Stage with Large First Stage for Number of Gates Greater Than Optimum for Fanout

can be slowed down without affecting the delay.

This optimization can be done by finding chains of gates off the critical path and setting the delay to the equivalent critical path segment length. The delays are made larger by the ratio of the desired delay to the current delay. The gates become smaller as do their capacitances. So a gate that drives a chain that is made slower can be made smaller and keep the same delay.

This part of the algorithm deals with delay: the desired delay of a chain and the delays of individual gates in the chain. After all desired delays are set, the transistor sizes can be set to meet the delay requests.

2.7.3. Heuristic Performance Optimization Can Slow Down a Chain of Gates

There are two rather counterintuitive cases that arise as a result of the way the optimization algorithms are applied. Performance optimization may actually make a circuit slower, and power optimization may make it faster.

There are cases where the performance optimizer makes a chain of gates slower than the original. This is rather obvious, since it does not give the absolute minimum delay solution. If the algorithm is run on a chain that has been optimized for truly minimum delay, the result will be slower, as can be seen in figure 2.10.

2.7.4. Power Optimization Can Speed Up a Chain of Gates

Another offshoot of the non-optimal nature of the delay optimizer is that the power optimizer can make a chain of gates faster. A situation in which this could happen is shown schematically in figure 2.11 in which each bubble represents a gate. The lower path is the critical path. The gate on the upper path will be off the critical path, so it will be made slower. The smaller gate will have less capacitance, so the gate that fans out to it can be made smaller with no change in delay. All the other gates in the chain can be smaller also.

If we look at the chain from the start to the fanout gate, then, we see that the load capacitance on the end of the chain has been reduced. If the chain was very long compared to the optimal number of gates in the chain, then the new graph of stage-versus-log(width), seen in figure 2.12, will show more gates at minimum size, along the horizontal part of the curve. Since these gates run faster than gates that must drive larger loads, they run faster, so the entire chain runs faster. If the chain is along the critical path, as this

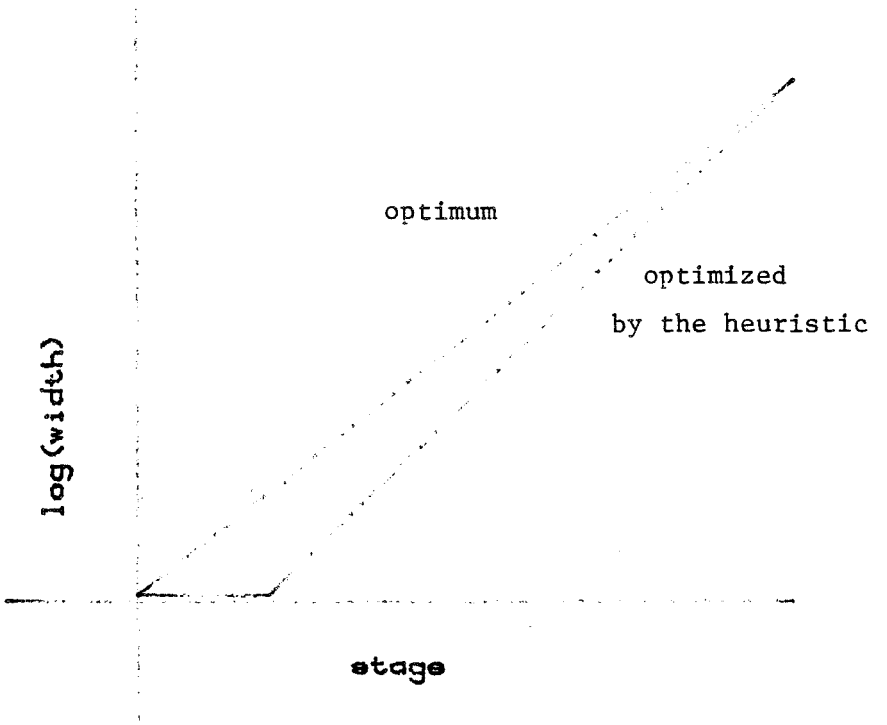


Figure 2.10. Delay Optimization of a Minimum-Delay Chain Makes it Slower.

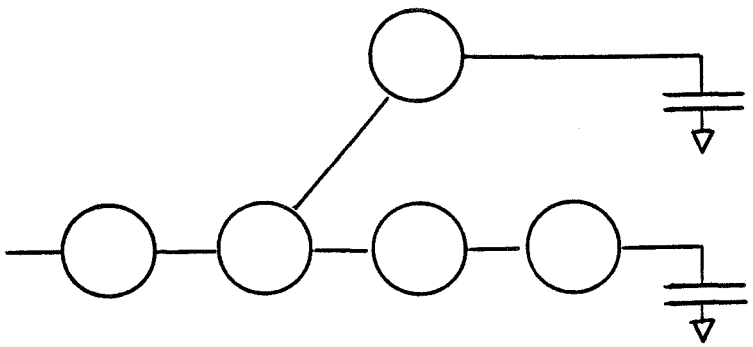


Figure 2.11. Situation in Which Power Optimization Can Reduce Delay

one is, the critical delay path will be shortened. Not all such situations will shorten the actual delay of the cell, since the critical path may shrink to become non-critical.

Speeding up of the chain because of lower capacitance in a chain occurs because we have more fast gates and fewer doing fanout. If we had a performance optimization algorithm that always produced the minimum delay solution, the straight line in figure 2.12, then we would still observe some cases where power optimization reduced delay, because the chain would still have less capacitance to drive, and therefore the fanout delays would be smaller.

2.8. Comparison of the Heuristic with Optimum Delay Results

This section deals with the adequacy of the heuristic delay algorithm. The delay of a chain of gates sized with the heuristic is derived and compared

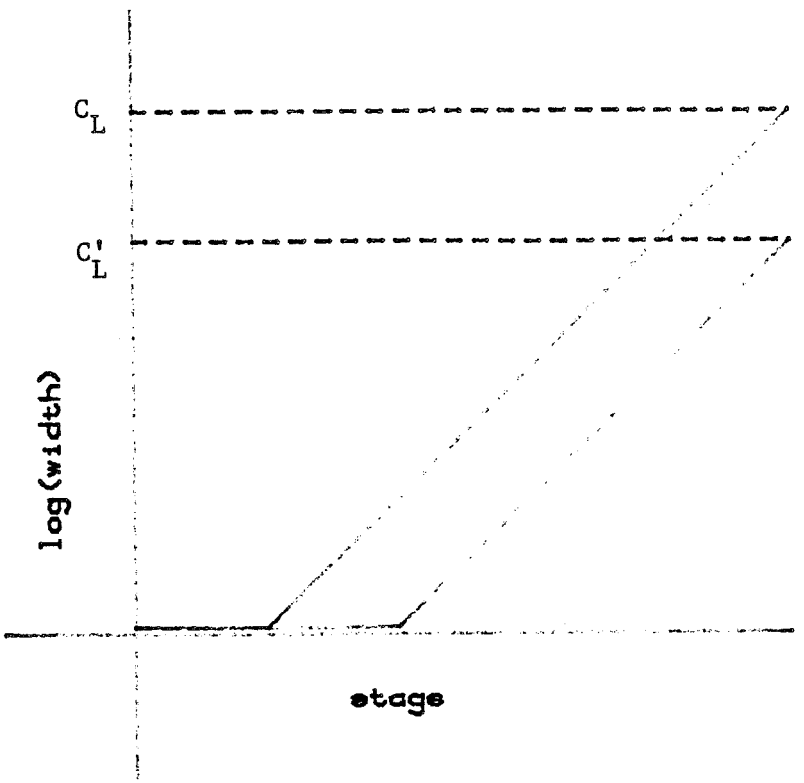


Figure 2.12. Power Savings Along Non-Critical Paths Can Shorten Delays.

with the optimum delay. The results indicate that the delay of a chain sized with the heuristic algorithm are reasonably close to the optimum.

2.8.1. Delays Using the Heuristic Sizing Algorithm

The delay of a chain of gates sized with the heuristic sizing algorithm can be derived for comparison to the delay for the optimum solution. The heuristic sizing algorithm works by sizing later gates in the chain to smoothly ramp up to drive the load on the output. One of the assumptions of the system is that there will be enough gates in the chain so there is no catastrophic jump at the start of the chain. So we concern ourselves with the chain with widths

shown in the graph in figure 2.7, which was discussed in section 2.7.

In this case, the chain can be divided into two pieces, the horizontal part in which all gates are minimum size, and the diagonal part, in which each gate is larger than the previous one by the fanout factor. By definition, the first gate that drives a larger load is the b^{th} gate in the chain. The delay of the chain is then:

$$D_{Heuristic} = \sum_{i=0}^{b-1} K + \sum_{i=b}^{N} fK = bK+(N-b+1)fK = K\,(b+f\,(N-b+1))$$

where $0 \le b \le N+1$. But, because the ramp is calculated from the output capacitance, b is not independent of the initial conditions. We know that:

$$w_0 f^{N-b+1} = w_{N+1}$$

Letting r_w equal $\frac{w_{N+1}}{w_0}$; and solving for b, we find:

$$b = N+1-\frac{\ln(r_w)}{\ln(f)}$$

Combining the two equations gives the delay of the heuristic as a function of the fanout factor, the length of the chain, and the ratio of the sizes of the transistors at the start and at the end of the chain:

$$D_{Heuristic} = K\left[N+1+(f-1)\frac{\ln(r_w)}{\ln(f)}\right]$$

This equation is not exact for two reasons. First, b must be an integer, since there can only be an integral number of gates. Second, the $b+1^{st}$ gate may not be larger than the b^{th} gate by f, since w_{N+1} may not be greater than w_0 by an even power of f. The result is that these two effects are opposite in

effect and nearly cancel because the single gate driving a smaller load is almost the same delay as two fractional gates, one driving no fanout, one driving the full fanout. This fractional-gate model is a little pessimistic, because it models the delay of the b^{th} gate the same way the heuristic model models a chain: instead of a straight line curve, it is modeled as a ramp on the output. This difference is minor in this case since it is less than one gate delay.

2.8.2. Comparison of the Heuristic Delay with the Optimum Delay

We wish to know for what relationship between the output load and fanout factor the heuristic gives the worst results. Therefore, we can form the ratio of the heuristic delay and the optimum delay and differentiate.

$$\frac{D_{Heuristic}}{D_{OPT}} = D_{ratio} = \frac{K\left[N+1+(f-1)\frac{\ln(r_w)}{\ln(f)}\right]}{K(N+1)r_w^{1/N+1}}$$

Since we wish to know the behavior as the ratio of the output load to the input load varies, we differentiate with respect to r_w, which was previously defined to be $\frac{w_{N+1}}{w_0}$:

$$\frac{\partial D_{ratio}}{\partial r_w} = \frac{r_w^{-\left[\frac{1}{N+1}+1\right]}}{N+1}\left[-\left(\frac{f-1}{\ln(f)}\right)\left(1-\frac{\ln(r_w)}{N+1}\right)-1\right]$$

Setting this equal to zero and solving for r_w, we get:

$$r_w = e^{(N+1)\left[1-\frac{\ln(f)}{1-f}\right]}$$

This equation gives the value of r_w at which the algorithm performs most

poorly for a given N and f. The optimal values, of course, occur when r_w equals 1 or f^{N+1}. In both these cases, the heuristic delay is equal to the optimal. This value of r_w can now be substituted back into the heuristic delay equation and the optimum delay equation to compare the delays in the worst case.

$$D_{Heuristic} = K\left[N+1+(f-1)\frac{\ln(r_w)}{\ln(f)}\right]$$

$$= K\left[N+1+(N+1)\left[\frac{f-1}{ln(f)}-1\right]\right]$$

$$= K\,(N+1)\left[\frac{f-1}{ln(f)}\right]$$

$$D_{OPT} = K\,(N+1)r_w^{1/N+1}$$

$$= K\,(N+1)e^{1-\left[\frac{ln(f)}{1-f}\right]}$$

These delay equations differ only in the final factor, which is dependent on f only. That factor can be evaluated for typical values of f, for this worst-case r_w. The results are shown in the table in figure 2.13. These values are within a factor of two, and for a fanout factor of 4, a typical value, the difference is approximately 25%. The worst-case of the heuristic gives delays that are comparable to the optimum delay for the simple chain.

Fanout Factor	Minimum Delay	Heuristic Delay	Percent Difference
2	1.359	1.443	6.2%
2.72	1.519	1.718	13.1%
4	1.712	2.164	26.4%
8	2.020	3.366	66.7%

Figure 2.13. Worst-Case Delay Factors for Typical Fanout Factor Values

2.9. Calculation of True Minimum Delay for a Chain with Capacitive Wires

The equations examined so far did not take into account parasitic capacitances in the wires between the gates. These parasitic capacitances are frequently as important as the gate capacitances. The chain we wish to solve, then is shown in figure 2.14, where there are additional capacitances between the gates:

$$C_i^{out} = C_{i+1}^{in} + c_i$$

The delay of the i^{th} gate, then, is:

$$D_i = kR_i(C_{i+1}^{in} + c_i) = k_d\left[\frac{l_0}{w_i}\right](l_0 w_{i+1} + c_i)$$

and the total delay for the chain is:

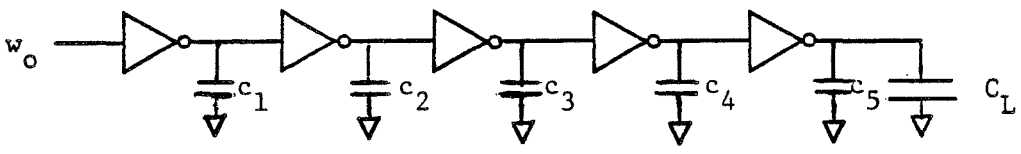


Figure 2.14. A Chain of Gates With Parasitic Capacitances.

$$D_{TOT} = k_d \sum_{i=0}^{N} \left(\frac{l_0}{w_i} \right) (l_0 w_{i+1} + c_i)$$

We can now differentiate this equation with respect to w_i to find the optimum.

$$\frac{\partial D_{TOT}}{\partial w_i} = -kd \frac{l_0}{{w_i}^2} \left(c_i + l_0 w_{i+1} \right) + k_p \frac{l_0^2}{w_{i-1}}$$

Setting to zero and solving for the ratio of the widths of two gates:

$$\frac{w_i}{w_{i-1}} = \frac{w_{i+1}}{w_i} + \frac{c_i}{l_0 w_i}$$

So the delay of the i^{th} gate is:

$$D_i = k_d \frac{l_0 c_i}{w_i} + k_d l_0^2 \frac{w_{i+1}}{w_i} = k_d \frac{l_0 c_i}{w_i} + k_d l_0^2 \left(\frac{w_i}{w_{i-1}} - \frac{c_i}{l_0 w_i} \right) = k_d l_0^2 \frac{w_i}{w_{i-1}}$$

There are two interesting aspects to this equation. First, note that later gates in the chain are slower than earlier gates in the chain. Earlier we saw that in order to make a lower-power chain, the later, larger gates should also be slower to save power. Second, the equation simplifies considerably in the minimum-delay solution. It states that the delay of the i^{th} gate should be the same as the delay of the $i-1^{st}$ gate if the extra capacitance were ignored. This is not identical to the case discussed earlier in which there is no extra capacitance, since the widths of all transistors are different.

2.9.1. Relative Importance of Chain Load versus Parasitic Capacitance

The width ratio equation can be rewritten as:

$$\frac{w_i}{w_{i-1}} = \frac{w_{i+1} + c_i / l_0}{w_i}$$

In the equation above, c_i / l_0 can be viewed as the width of a transistor that would produce the parasitic capacitance c_i. Thus, parasitic capacitances can be modelled simply as a larger gate for each stage to drive. Alternatively, we can view the equation as a ratio of capacitances. In either view, the effect of parasitic capacitance on the sizing of a gate is equal to the effect of capacitance of the next gate in the chain.

The first term in the equation is the same as the term in the "pure" case earlier which ignored capacitance between the gates. It is the contribution of the ramp to satisfy the load on the end of the chain of gates. The second term represents the contribution of the parasitic capacitance on the output of this gate. The relative size of the two terms is determined by the relative capacitances of the next gate in the critical path and the parasitics on the output. If either term is much larger, we could safely ignore the other.

The capacitance per unit area of a wire is only an order of magnitude less than the gate capacitance. Therefore, even a short wire gives a capacitance comparable to that of a minimum-sized transistor. The capacitance of a long wire dominates the gate capacitance. So, the immediate parasitic term is as important as the ramp term and may completely swamp the ramp term for longer wires.

The parasitics on the output of a gate affect earlier gates in the chain because they appear as a larger transistor to drive, creating, from the point of view of the preceding gates, a larger ramp. The ramp effect diminishes exponentially, though, since each gate in the chain is sized to be smaller than the next one (even in the optimum case in which the fanout factor is

chosen to be the optimum number).

It is possible to solve the equations above for the optimum sizes of all gates in a chain with parasitic capacitance, but it is not easy. The width ratio equation is quadratic, the simple case of two gates is quartic. Longer chains of gates give higher power polynomials to solve to find the sizes of the gates. They can be solved, but the approximation methods are not amenable to a fast-turnaround system. We have seen that the parasitics are at least as important as the optimal ramp term. The heuristic algorithm presented in later chapters uses the heuristic sizing method described earlier with the addition of parasitic capacitance, giving a simple, fast, and accurate optimization method.

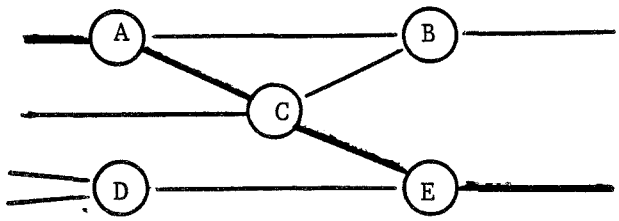


Figure 2.15. A Graph of Gates With a Critical Path.

2.9.2. Extension of Capacitive Chain to Graph-Like Gate Structure

Let us examine a slightly more complex case. Instead of a simple chain of gates, we have a graph-like structure, shown in figure 2.15. The equations above can be used the model this graph structure. Fan in to a gate is not a problem, since we consider a critical path, which ignores alternate non-critical paths.

The critical path is the chain under consideration. There is some fan out to gates off that critical path. These transistors appear as added parasitic capacitance that has nothing to do with the chain driving the load. The capacitance from gates off the critical path contribute to the c_i. A single gate fanout would make c_i comparable to the capacitances of the gates in the chain. In many applications, such as clock drivers and PLAs, fanout is very large, perhaps dozens of gates. In these cases, the term in the sizing equation dealing with the parasitic capacitances, the c_i, will dominate the term dealing with the load on the output of the critical path.

Gates will have large loads on them independent of the loads due to the later transistors in the chain. Therefore, optimal sizing of the chain is not essential. Much more important is the consideration of the effect of the parasitic capacitance in the calculation of the gate sizes. Parasitic capacitances due to wires and gates off the critical path are important. These effects are hard for designers to address because they are hard to measure and because many wires are added automatically by the design system.

The large loads in the chain imply that optimum sizing of gates in long chains driving large loads is not as important as sizing of single gates to drive the

parasitics on them. The heuristic performs well on a chain of gates and will perform equally well with parasitic capacitances included.

CHAPTER 3

Andy – A System That Optimizes Performance in Sticks Circuits

This chapter describes *Andy*, a program that takes a logical composition specification and performs the electrical composition, which involves three tasks. Most importantly, Andy improves the speed of the circuit. In addition, it ensures proper pullup-pulldown ratios on all gates including those that have some inputs gated by pass transistors. Andy also flags the dangerous condition where the gate of a pass transistor has itself been gated by a pass transistor.

This chapter includes a description of the program and its environment and gives a user's view of the optimization algorithms. The details of the algorithms are given in a later chapter.

3.1. Overview of Andy

Andy is a program that optimizes delays in circuits that are defined in a symbolic notation, the Sticks Standard. The Sticks Standard and its terminology is described in detail in Appendix B. The interface to the optimizations is the major facility in Andy. The optimizations can be run independently or as a group and the user may view the result or get statistics on the resulting circuit.

Besides an interface to the performance and power optimization algorithms, Andy has several utility functions for altering Sticks cells, to prepare the

design for the optimization, or to direct the optimizations. These utilities add parameters to connectors on the edges of cells, and add constraints on Sticks components and interconnecting twigs in Sticks Standard cells. Andy has no Sticks editing facilities. Changes in the circuit must be done with some other tool.

Andy is made up of more than 5000 lines of Simula code, not including the shared graphics package (another 6000 lines). Associated utilities involve another 5000 lines. The Andy compiled code takes 107K words in the DEC-20, and has 228K words of data space. Although Andy keeps all data in memory, this space is adequate for small and medium-sized examples. Full-scale large chip optimizations would require use of disk storage to avoid filling memory.

Andy fits in the current design system as shown in figure 3.1. Andy reads Sticks Standard files [Trimberger 1980a] which can be made with Rest [Mosteller 1981], Riot [Trimberger 1982b], or other Sticks tools. Unlike some tools, Andy accepts Sticks files that describe the entire design hierarchy.

Andy is a command-oriented design aid. Andy reads Sticks files and processes them according to commands by the user. When the user is content with the design, he may write it back in Sticks form.

3.2. Commands and Capabilities

This section provides an introduction to Andy. It is basically a summary of the commands that can be given to the program grouped by function. For a complete description of Andy, see the Andy User's Manual in Appendix A.

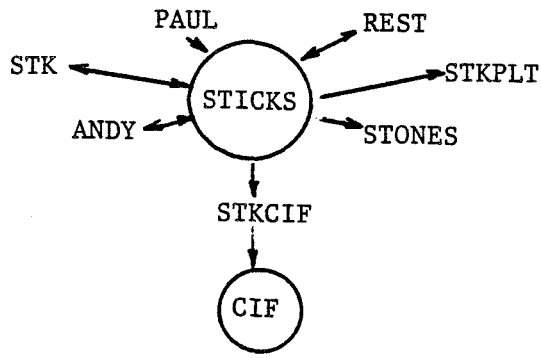


Figure 3.1. Andy in the Caltech Design World

3.2.1. Input and Output

Andy reads and writes the enhanced Sticks Standard. In addition, Andy can write a dump of its internal form including the node and gate information that was derived from the Sticks. A complete description of the additions to the original Sticks Standard required for Andy is given later in this chapter.

3.2.2. Cell Management

In an interactive system such as this one, cell management facilities are required to help the user select the cells to be optimized. Therefore, Andy has facilities for listing cell names, entering a cell to view the cells defined within it, and clearing the list of cells.

3.2.3. Plotting

It is often necessary to view the data to understand what the optimization

has done or to identify the places where the design should be modified so more optimization can take place. Andy has a complete plotting package that includes cell selection, windowing, output device selection and scaling of the plot.

There are options on plotting that enable the user to plot only the cell bounding box and connectors, and to optionally include component names on the plots.

The user may plot the cell as a symbolic Stick diagram or as an abstract gate diagram, showing the connections from the connectors on the cell and the connections between gates.

3.2.4. Stick Modification Utilities

There are two major alterations that a user must perform on the Sticks data in Andy. First, connectors must be labelled with types and given default loading. Second, constraints must be added to limit the optimization process. Constraints include loading constraints and transistor size constraints. The types and constraints are described later in this chapter. That section also includes an example of their use and an explanation of their necessity.

These constraints can be expressed textually, if the name of the component is known. This textual specification may not be easy if the Sticks cell was generated automatically, so Andy also provides a graphical means of identifying components. After the cell has been plotted one can point to components and set the name, connector loading, and transistor length and width. Also, constraints can be made on components. Bad constraints can

be removed.

3.2.5. Parameters to the Optimizations

The delay and power optimizations use several global values for critical parameters. the user may set these values and thereby direct the overall operation of the optimization algorithms. The user may turn off and on the inclusion of capacitance on wires. The wire capacitance is usually on, because it is a significant load in most circuits. Examples in the following chapter show a significant difference in device sizes when wire capacitance is included.

In a further guidance of the optimization, the user may control whether or not CLOCK nodes on pass transistors will break paths during delay calculation and power optimization. Turning it on allows optimization for minimum clock cycle, turning it off allows optimization for minimum delay through a pipelined processor.

The user may adjust the most important number in the performance optimization, the fanout factor. The fanout factor is the number of minimum transistor capacitances that should be driven by a minimum transistor. The fanout factor says in some sense how concerned the user is with power versus delay. Larger fanout factor means larger delay but lower power. It may be set to any value greater than one, and is set initially to four.

A third parameter is the default loading on a connector. Is is usually not reasonable that connections to the outside world have no capacitance on them. It is possible to put a specific load on a specific connector, and it is also possible to put a default load on all other connectors.

3.2.6. Statistics

To help the user determine the quality of a design, Andy reports statistics on the cell. The user can get the delay of the critical path, a listing of the critical path, the power consumption of the chip and the product of the delay and power. The delay and power estimates from Andy are not exact because constants are ignored, and they are based on the simplified models described earlier, but one set of statistics can be compared to another to get an idea of the relative goodness of two designs. The following is an example of Andy output:

```
Cell PLA.  Delay: 9.64E-01.  Power: 2.34E+01.  D*P (unscaled): 2.25E+01

Critical Path for cell PLA C:Y1IN G:INBUF_P1C4 G:INBUF_P2C4 G:AND_P5 G:OR_P6
 G:OUTBUF_PU1C5.  Delay: 9.64E-01
Critical path changed.
```

3.2.7. Constructing the Data Structure

Andy has commands specifically to build the data structure. The data structure must be built before the optimization steps, so the optimizations build the structure if necessary. These commands to separately generate the nodes and recognize the gates is included primarily as a debugging tool.

3.2.8. Delay and Power Optimization

Delay and power optimization are Andy's main tasks. They can be performed separately or sequentially with a single command. Separate commands for each step are provided more as a debugging aid than as a user feature, but there may be some situations where one or the other is not desired. The

delay and power optimization is described in more detail later in this chapter.

3.2.9. Area Optimization

Delay and power optimization change device sizes and may cause design rule violations, mandating that area optimization be performed on the cell. Andy sends simple cells to Rest to do this optimization. Rest cannot currently handle cells with hierarchy, so some other software is needed for dealing with area optimization of composition cells. An associated program, STK, can be used to remove the hierarchy so Rest can optimize area.

3.2.10. Debugging Aids

There are a few commands of no interest to users which generate trace information during the data structure construction and during the optimization stages. There is also a command in Andy to enter the SIMULA debugger for further examination of the internal structure of the program.

3.3. Input Requirements

Andy reads Sticks Standard format [Trimberger 1980a]. A sample Sticks Standard cell is shown in figure 3.2 and a drawing of the cell in figure 3.3. The Sticks form describes components, such as transistors, resistors, contacts and connectors; twigs, which are interconnection; and constraints, limits on the cleverness of the optimizing program that will optimize the data.

```
CELL srcell 250 4
  COMPONENTS
    CONNECTOR T GROUND: gndl -48 -45  gndr 48 -45 ;
    CONNECTOR T INPUT: in -48 -29 ;
    CONNECTOR T POWER: vddl -48 45  vddr 48 45 ;
    CONNECTOR T OUTPUT: out 48 -29 ;
    CONNECTOR T CLOCK: clktop 8 59  clkbot 8 -59 ;
    NENH W 16 L 8: pd -20 -29 ;
    NENH W 8 L 8: ps N 0 -1 8 -7 ;
    NRES W 8 L 32: pu -20 1 ;
    NBUT: but N -1 0 28 -15 ;
    NDM: N1 -20 -45 ;
    NDM: N3 -20 45 ;
  TWIGS
    POLY(8):=   clkbot   8,-43   ps.G1   clktop;
    METAL(12):=   gndl   N1   gndr;
    DIFFUSION(8):=   N1   pd.SOURCE;
    POLY(8):=   in   pd.G1;
    DIFFUSION(8):=   pd.DRAIN   pu.DSOURCE   ps.SOURCE;
    POLY(8):=   28,-29  (   out)   but.P;
    DIFFUSION(8):=   pu.DRAIN   N3;
    DIFFUSION(8):=   ps.DRAIN   but.D;
    METAL(12):=   vddl   N3   vddr;
  CONSTRAINTS
    in.Y=out.Y;
END

CELL sr 250 4
  COMPONENTS
    srcell :   sr1 48 0;
    srcell :   sr2 144 0;
    CONNECTOR T GROUND: gndin 0 -45  gndout 192 -45 ;
    CONNECTOR T POWER: pwrin 0 45  pwrout 192 45 ;
    CONNECTOR T INPUT: input 0 -29 ;
    CONNECTOR T OUTPUT C 10: output 192 -29 ;
    CONNECTOR T CLOCK: clktop1 56 59  clkbot1 56 -59 ;
    CONNECTOR T CLOCK: clktop2 152 59  clkbot2 152 -59 ;
  TWIGS
    Metal :  =  sr1.gndr sr2.gndl;
    Metal :  =  sr1.vddr sr2.vddl;
    Poly :  =  sr1.out sr2.in;
    Metal :  =  pwrin sr1.vddl;
    Metal :  =  pwrout sr2.vddr;
    Metal :  =  gndin sr1.gndl;
    Metal :  =  gndout sr2.gndr;
    Poly :  =  input sr1.in;
    Poly :  =  output sr2.out;
    Poly :  =  sr1.clktop clktop1;
    Poly :  =  sr2.clktop clktop2;
    Poly :  =  sr1.clkbot clkbot1;
    Poly :  =  sr2.clkbot clkbot2;
  CONSTRAINTS
END
```

Figure 3.2. The Sticks Standard Representation of a Shift Register Segment

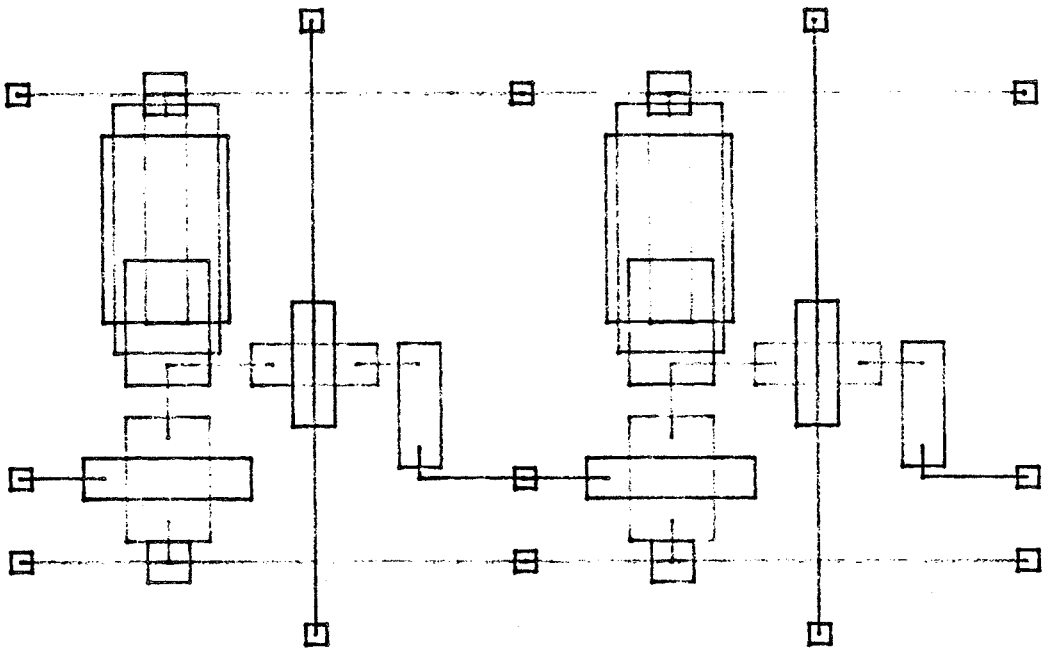


Figure 3.3. The Shift Register Segment from Figure 3.2

3.3.1. Parameters

Unaugmented Sticks Standard does not include enough information for performance optimization. Therefore, several parameters on components and constraints were added to facilitate the performance optimization. New parameters on components are shown in figure 3.4.

The gate finding algorithm can find pass transistors most of the time. However, there are some circuits that confuse it. By explicitly declaring the pass transistors in these cases, the gate finding algorithm will succeed and performance optimization will produce better results.

The type of a connector is vital to the device recognition and performance

On a Transistor	
P	The transistor is forced to be a pass transistor.
On a Connector	
T <type>	The type of a signal on the connector.
C <number>	A default capacitance on the connector.
O <number>	A default capacitance on the connector.
P	The signal on the connector came under a pass transistor.

Figure 3.4. Table of Additional Parameters on Sticks Components

optimization algorithms. The types understood by Andy are shown in the table in figure 3.5.

The only required types are POWER, GROUND, INPUT and OUTPUT. Unlabelled connectors are assumed to be IO. OUTPUT and IO connectors may have an additional parameter to simulate a load of a given number of minimum-sized transistors on the output. This simulated load is used when the cell is not used as an instance in a larger circuit so there is no real load on the connector.

For delay calculation, every INPUT to a cell is assumed to be driven by a gate that is smaller than its load by the fanout ratio, or by a minimum size transistor, whichever is larger. Also, an input connector is assumed to represent a restored logic signal unless it is marked that it came under pass transistor. Connector types, capacitances and unrestored signal markings

POWER	Power connection from the power supply.
GROUND	Ground connection from power supply.
INPUT	Signal generated outside the cell driving logic inside the cell.
OUTPUT	Signal generated inside this cell driving logic outside the cell.
IO	Signal that acts as both INPUT and OUTPUT.
BUS	Functionally equivalent to IO.
CLOCK	Signal that delimits ends of time phases.

Figure 3.5. Table of Connector Types Used in Sticks Standard.

are only used on connectors on the cell on which the performance optimization is being done. Connectors on instances in the hierarchy are absorbed in the node merging step described in the next section.

3.3.2. Constraints

Andy uses some additional constraints beyond the simple geometrical constraints described in the Sticks Standard document. These constraints limit the performance optimizer, and are summarized in the table in figure 3.6.

Andy modifies transistor lengths and widths, therefore the user has the option to restrict that resizing on specific transistors. A pre-defined capacitance that is applied to a twig is transferred to the node that includes the twig when the node creation is done. This constrained capacitance then takes precedence over the capacitance that is calculated for the node. This capacitance constraint is useful in shared bus situations where the designer knows that each driver need not drive all loads off the bus at once. The performance optimizer assumes the worst, looking through pass transistors pessimistically, unless the node capacitance is constrained.

The gate finding algorithm terminates at a BUS node. Andy's gate recognition algorithm will follow nodes to GROUND, which is incorrect in many cases with

trans	.L = <number>	The length of a transistor.
trans	.W = <number>	The width of a transistor.
twig	.C = <number>	A pre-defined load capacitance on a twig.
twig	.B	The twig referenced is on a BUS-type node.

Figure 3.6. Table of Additional Sticks Standard Constraints.

shared busses. The BUS constraint on a twig will cause the node that contains the twig to be a BUS node, limiting the gate recognition algorithm.

Improper use of these constraints can cause the performance optimization to give wildly inaccurate results, so they should be used sparingly.

3.4. The Data Structure

The circuit is made up of gates that drive capacitive loads on electrical nodes. A node is a collection of all the Sticks twigs and component references that are always at the same electrical potential (after everything settles down). Nodes may cross the boundaries of the physical hierarchy. The derivation of the nodes for a simple cell is shown in figure 3.7.

3.4.1. Nodes

Every node has pointers to all Sticks components and twigs attached to the node. Elements in the node are separated into two categories: those that drive the node, *drivers*, and those that are driven by the node, *loads*. The distinction is made so much of the gate recognition and optimization can run faster. Twigs in the node are always loads. Transistor-like devices that drive the node have either the source or the drain of the device attached to the node. Devices that are driven have the gate attached to the node. A transistor may be both a driver and a load if its gate and either source or drain are connected to the node. Depletion-mode pullup devices are treated separately as resistors. Pass gate transistors, which are recognized in the gate derivation are both drivers and loads on the node.

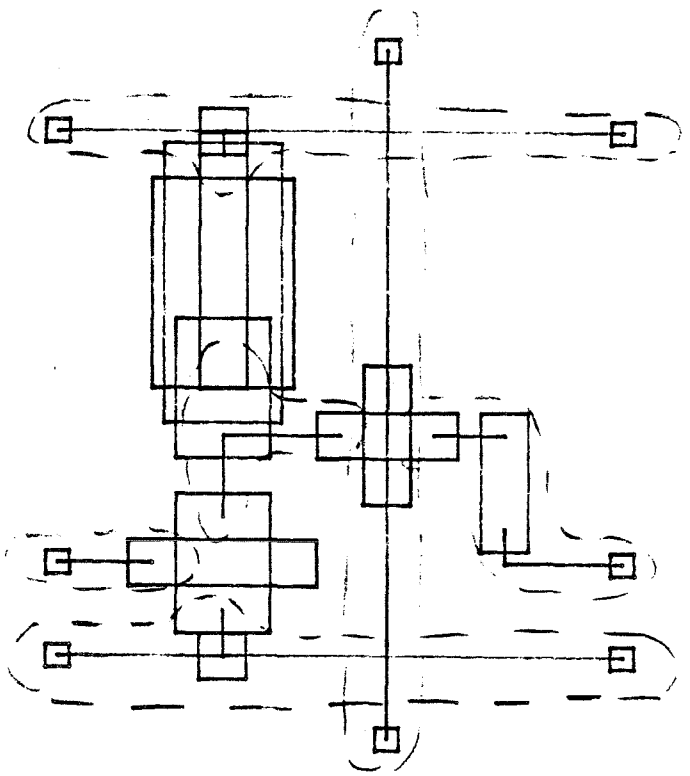


Figure 3.7. The Node Derivation of a Simple Cell.

3.4.2. Gates

Once the node structure is derived, it is followed to extract gates. Gates are recognized on the entire cell submitted for optimization. The algorithm follows nodes across cell boundaries if necessary and moves up and down the design hierarchy to extract the gate information.

In nMOS circuits, there are basically two kinds of gates: *restoring logic gates*, with a pullup device and a pulldown structure, and *transmission gates*, which are pass transistors (figure 3.8). The former are unidirectional and are the form most often envisioned as gates in circuits. These unidirectional gates

are made up of a single pullup device connected to the POWER node on one side and the output node on the other, and a tree-like pulldown structure connected between ground and the output node. A transmission gate is formed by a transistor that does not connect directly to POWER and does not connect even indirectly to GROUND. This is the same distinction used in the gate extraction algorithm for the MOTIS simulator [Chawla 1975].

The gate recognition algorithm distinguishes between restoring logic gates and transmission gates. However, there are some MOS structures that are

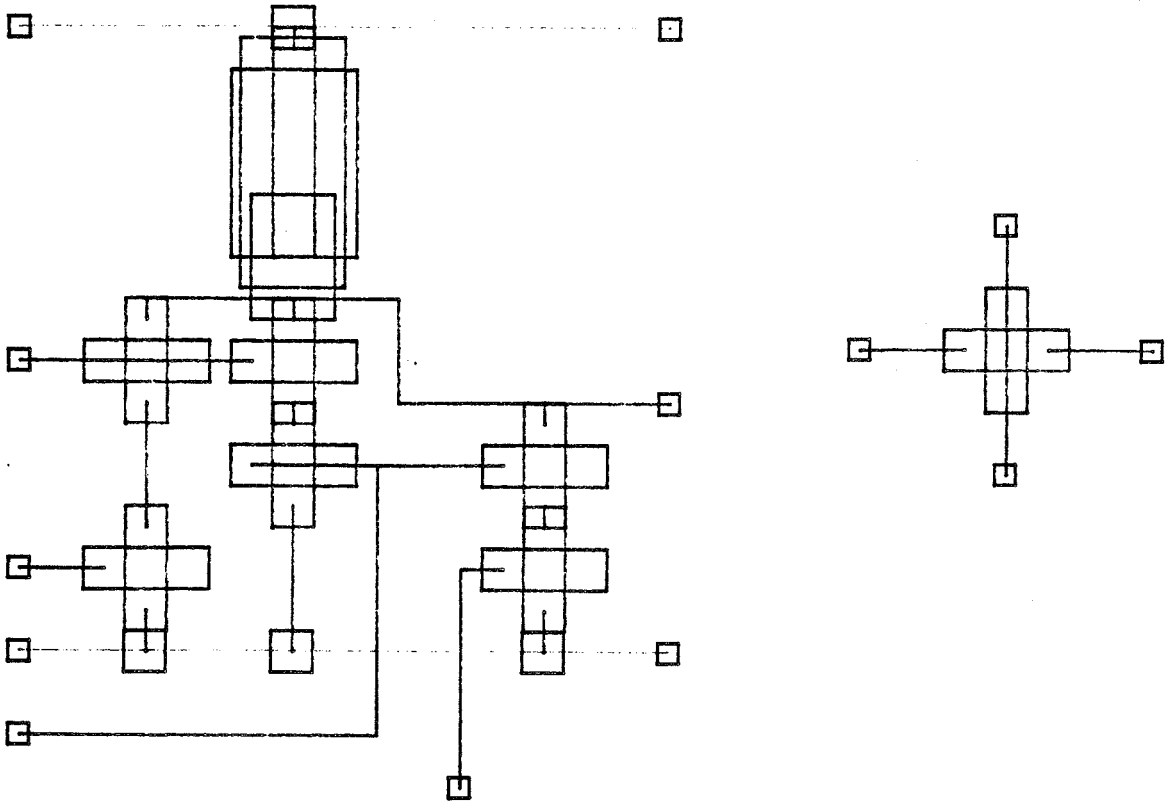


Figure 3.8. Types of Gates. a) Restoring Logic Gate. b) Transmission Gate

not allowed, and some that will may not result in a gate derivation that the designer wished. Gates may have only one pullup and one output. The pulldown structure must be a true tree structure with no internal connections. Examples of well formed gates are given in figure 3.8, and ill-formed gates in figure 3.9.

The resulting data structure is shown in figure 3.10. Gates drive their output nodes. Nodes drive transistors. Transmission gates are accessed via the nodes on either side.

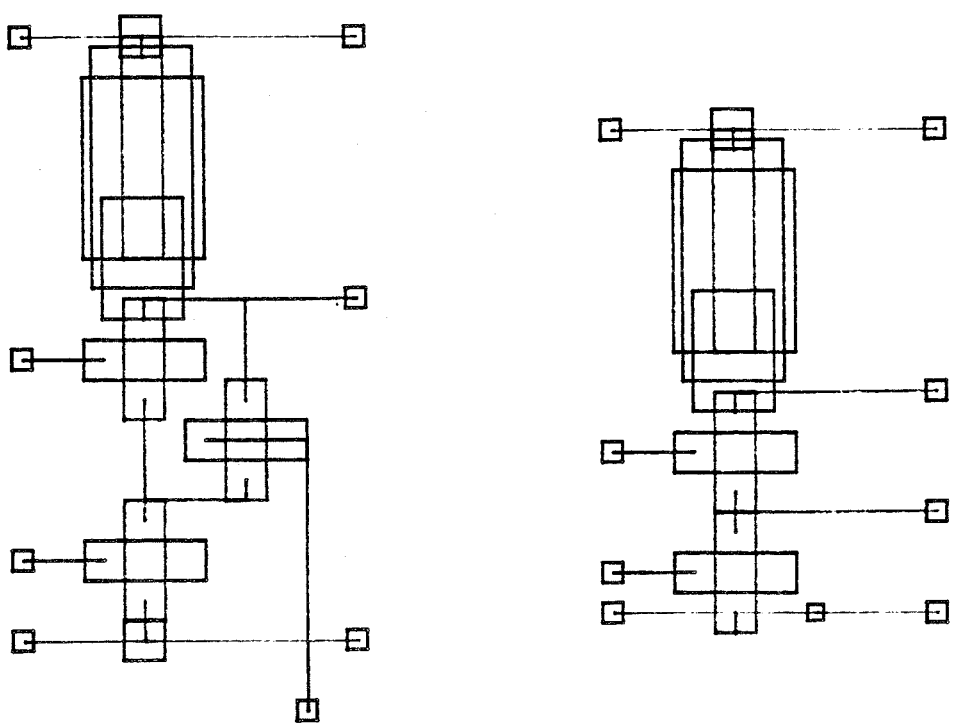


Figure 3.9. Ill-formed Gates

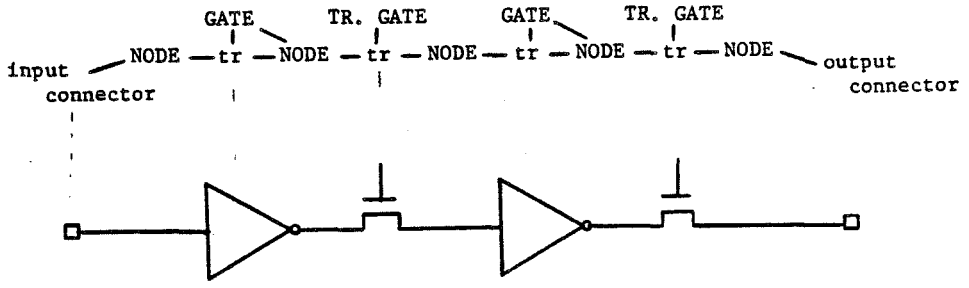


Figure 3.10. The Node and Gate Data Structure.

3.4.3. Performance Optimization Design Rules

Performance optimization can be expressed in a somewhat formal manner by defining "design rules" that the algorithm enforces and attempts to meet as closely as possible. These rules are presented as a means of explanation of the function of Andy, not as a description of the algorithm. Andy is not a "rule based" system in the artificial intelligence sense. Design rules obeyed by Andy are:

(1) The minimum transistor width is 2 lambda. Minimum transistor length is 2 lambda.

This rule sets the minimum gate dimensions, which determine the cutoff for making transistors smaller. These dimensions also determine when the algorithm optimizes devices by changing width rather than changing length of transistors.

(2) A pulldown structure in a gate must have at most one square transistor resistance for each *<fanout>* minimum transistor sizes of gate capacitance that are driven by the gate.

(3) A pullup resistor must have at most one quarter square depletion transistor resistance for each *<fanout>* minimum transistor sizes of gate capacitance that are driven by the pullup.

These rules comprise the gate fanout rule. Meeting these rules is the main task of the performance optimizer. No gate may drive more fanout than the fanout variable allows. As discussed earlier, optimal delay occurs when this number is *e*, but it is usually between four and eight. In the Andy system, the default value is four, but it may be changed by the user. The fanout number must always be greater than one.

Rule 2 makes a statement about the pulldown structure of the gate, not about individual transistors. Therefore, to obey this rule, the gate structure must be determined to size the devices.

(4) A pullup device that is not a depletion-mode transistor with the gate tied to the source indicates that the gate driving current is four times that of a normal gate.

A transistor-like pullup must be either a precharge device or a super-buffer device. Either way, the pulldown becomes the limiting resistance in the gate. Therefore, the gate can drive four times as much load as a normal gate.

(5) A pass transistor must have at most one quarter square gate resistance for each *<fanout>* minimum transistor sizes of gate capacitance that are driven through the pass gate.

This is the pass transistor sizing rule. It makes pass transistors the same resistance as a pullup resistor. This heuristic is included so neither the pass transistor nor the pullup resistor is the dominant resistance on the signal.

(6) Transistor gate resistances and capacitances and interconnect capacitances are assumed to be:

Transistor Capacitance	4.0×10^{-4} pf/μm^2.
Diffusion Capacitance	1.0×10^{-4} pf/μm^2.
Polysilicon Capacitance	0.4×10^{-4} pf/μm^2.
Metal Capacitance	0.3×10^{-4} pf/μm^2.
Transistor Resistance	1.0×10^{4} Ω/■.
Wire Resistance	0.0 Ω/■.

The resistances and capacitances of the elements of the design are used by the performance optimization. These capacitance numbers are taken from [Mead 1980]. The precise values of these numbers are not important, but their ratios are important, particularly the relative sizes of the capacitances for transistors and interconnect.

(7) The resistance of a transistor that has had the signal on its gate go under a pass transistor should be considered double.

This rule compensates for the lower gate voltage on the transistors driven by signals that have gone under pass transistors. The gates will be made wider.

(8) The maximum length of a pulldown is 2 lambda.

This rule places an upper limit on the resistance of the pulldown and therefore an upper limit on its delay. This keeps the power optimization from going overboard when saving power on paths that are very far off the critical path.

These rules define an optimum delay that is not a true global optimum. The result will be a local optimum, subject to the constraints supplied by the system, the accuracy of the design rules and the model of integrated circuit performance. This is in the same sense that symbolic layout compaction achieves a local optimum, subject the the constraints of design rules and algorithmic limitations.

3.5. Optimization Overview

This section contains an outline of the performance optimization algorithm used in Andy. The algorithm can handle constraints on transistor sizes and loading. Figure 3.11 shows the optimization algorithm block diagram. First, Andy reads a Sticks file and extracts the node and gate data structure. Then performance optimization is done followed by the power optimization step. In the end, Andy writes a Sticks Standard file.

The delay and power optimization in Andy is a purely electronic method, dealing only with the electrical capacitive attributes of the circuit. Andy optimizes performance of an integrated system by altering device sizes to match the loads on them. Andy also makes proper pullup/pulldown ratios and fixes gate ratios for gates whose inputs went under pass transistors.

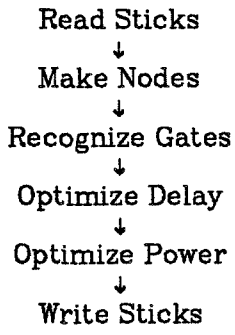


Figure 3.11. Performance Optimization Flowchart

Proper ratios are a side-effect of the gate sizing algorithm.

There are many other methods of performance enhancement that could be used: wires could be shortened, logic stages could be inserted or deleted to make the fanout factor as close to optimum as possible, duplicate logic could be introduced to avoid fanout. These changes are considered design issues to be handled by the designer, as opposed to layout issues that are handled by the design system. The output from Andy will direct the designer to make these kinds modifications of the logic to further improve the performance.

The optimization task is divided into two separate operations, speeding up all gates to optimum speed, then slowing down all gates off the critical path to save power and area. Gates that are off the critical path may be slowed down so that all paths have delays equal to the delay of the critical path. The slower gates consume less power.

Delay optimization of all gates is necessary to determine the delay of the critical path, since the critical path is not known until the minimum delay has been determined. The gates on the critical path must be made optimum size, and optimizing the other gates gives initial device sizes that are less

critical than the critical path. Therefore, when gates off the critical path are slowed-down to save power, the amount of slack delay is known.

3.5.1. Delay Models

The delay of a restoring logic gate is proportional to the resistance (R) of the pullup times the capacitance (C) on the output node. The capacitance may include the parasitic capacitance on the wires. The delay through a chain of gates is the sum of the RC delays. This RC delay is the measure used in estimating delays in the optimization algorithms. The amount of power dissipated by these gates is inversely proportional to the resistance of the pullup.

Transmission gates are potentially bidirectional, and current supplied elsewhere will pass through a pass transistor. The optimizer attempts to keep pass transistors from being serious detriments to the performance of the circuit. It is also unreasonable to make pass transistors have a negligible effect of performance at a large cost in area. Therefore, the pass transistor resistance is set to be the same as the resistance of a pullup that would have to drive the larger of the capacitances on each side of the gate. Pass transistors are not considered in the determination of the delays in a circuit except as an additional capacitance on the node, and since they have no connections to power and ground, they do not contribute to power consumption.

There are places of special concern with bus-like structures in which the signal goes through a pass gate. Logic on the other side of the pass gate may at some times require that the node drive logic, and at other times the logic

may drive the node. The algorithm assumes worst case in all pass transistor situations: it assumes that it may have to drive all logic past a pass transistor at once. Therefore, the capacitance on a node that runs to a pass transistor includes the capacitance of the transistor and the capacitance on the node on the other side of a pass transistor as well. The capacitance calculation goes through all pass transistors. To limit this, the user may constrain a capacitance on a node, such as the bus node.

Every circuit has some connections to the outside world that have some driving requirements. These requirements may be supplied as constraints on the loading of the node. For example, a bonding pad node may be constrained so it can drive three TTL loads. If they are not given explicitly, the default value, which the user can set, is used.

3.6. Performance Optimization

The performance optimization algorithm works as follows:

```
PROCEDURE optimize_performance;
WHILE some gates are yet to be sized DO BEGIN
    FOR all gates DO  IF gate.known_load THEN move_into_ready_list;
    IF no gates in ready list THEN move any gate into ready list;
    FOR all gates in ready list DO gate.setsize;
END
```

The transistor sizing algorithm maintains two lists of gates: gates that have not yet been sized and are ready to be sized, and gates that have not yet been sized but are not ready to be sized. A gate is ready to be sized when all the loads on its output node are known. Known loads are twig capacitance, output connectors, and transistor gate connections on transistors that have already been sized.

The gates in the former list are processed, setting the sizes of the transistors that make them up, depending on the load on the output node. Transistor sizes are set to *MAX(minsize, output node capacitance/fanout factor)*. When a gate is sized, it is removed from the list:

```
PROCEDURE gate.set_size;
BEGIN
    basicresistance := MAX(min_trans_size,
        const*output_capacitance/fanout factor);
    pullup.setresistance(basicresistance*longest_NAND_length*pullup_ratio);
    FOR all pulldowns DO BEGIN
        pulldown.setresistance(basicresistance);
        pulldown.driver_node.driver_gate.sized := FALSE;
    END;
    sized := TRUE;
END
```

When a transistor in a gate is sized, the gate that drives the node that drives the gate of the transistor is moved into the list of unsized gates, since its load has changed.

As transistor sizes are set, more nodes have known loads. The gates that drive these nodes nodes can then be sized and so forth. The algorithm proceeds backward from the circuit outputs through the circuit until all gates have been sized.

In a circuit with a feedback path, the loads on some gates are dependent on the size of their own transistors. These gates cannot be sized because none of the the loads on the output nodes is defined. The proper sizes of all the transistors in the loop can be found by simultaneously solving the device size equations. However, Andy solves these equations with much less computation by relaxation into a fixed point. Andy detects and breaks the loop by picking one gate arbitrarily and sizing it. The transistors in the sized gate are now known loads, so the gate before the chosen gate can be sized,

and so on. Eventually, the optimization makes its way around the loop to re-size the first gate. This re-sizing terminates when a transistor changes size by less than five percent. A transistor that does not change much does not move the driver of its gate node into the list of unsized gates.

3.7. Power Optimization

The power optimization algorithm can be expressed in general as follows:

```
PROCEDURE optimize_power;
BEGIN
    find_paths;
    sort paths into decreasing order;
    FOR all paths DO BEGIN
        find first gate that has not been optimized yet;
        current_delay := delay at end of the path -
            delay at first unoptimized gate;
        desired_delay := constrained delay at end of the path -
            constrained delay at first unsized gate;
        expand_ratio := desired_delay/current_delay;
        FOR all gates between first unsized gate and end of path DO BEGIN
            gate.constrained_delay := gate.current_delay * expand_ratio;
            gate.sized := TRUE;
        END;
    END;
    FOR all gates DO set delay to constrained delay;
END
```

Power optimization is done by sorting all the paths of gates in the cell into decreasing length. A path is a chain of gates that starts at the input connectors or at a pass transistor that is gated by a CLOCK node (if the clocking mode is turned on) and ends at the output connectors, at the input to a gate or at a pass transistor that is gated by a CLOCK nodes (if the clocking mode is turned on). The paths of a simple circuit are shown in figure 3.12.

Each path is treated independently in the power optimization. All gates

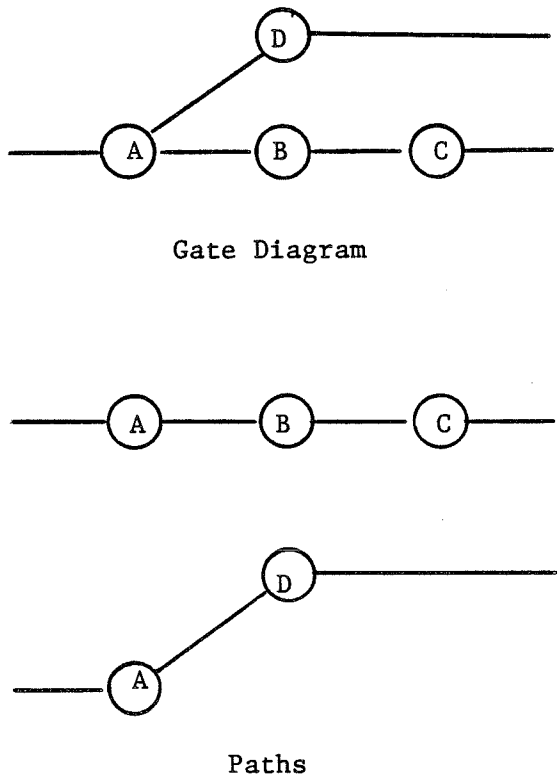


Figure 3.12. The Paths in a Simple Circuit.

along the beginning of the path that have already been sized with the performance optimizer are chopped off. The delay of the remaining gates is compared to the difference in delay from the beginning of the path (either the input connectors or the last gate that was chopped off) to the end of the path (the output connector or the gate at which the path stopped). All gates in the chain are made slower by the ratio between the desired delay and the current delay.

In the end, then, all path delays are as long as the longest delay. In accordance with the rules above, though, no gate is made so slow that a

pulldown transistor width is smaller than its length. So some paths may remain faster than the critical path.

The longest delay is usually the critical path delay, but it can be set by the user, so the delay of the entire cell can be set to a desired value by the power optimizer.

CHAPTER 4

Examples of the Andy Optimizer Operation

This chapter gives examples of the optimization in Andy. The first few examples are small cells, designed to give the reader a better understanding of the changes Andy makes to a cell. The larger examples in the later parts of this chapter are "real world" chips, in the sense that they perform some useful function and are adequate examples of the optimization that can be expected by using Andy on real designs.

Examples shown in the this chapter were prepared using Rest to make the cells and Riot to assemble them. Some small examples were made with Paul [Trimberger 1980b]. A special purpose Sticks PLA generator was used to make the PLAs. The Spice simulator [Cohen 1978] was used to generate some of the timing results. Other tools were used at various times to process the Sticks files.

4.1. Small Examples

These examples are included to show in some detail the effects of the delay optimization on a simple gate with different loads. This section includes some simulation results from Spice for comparison with Andy's statistics output.

4.1.1. A Simple Loaded Inverter

The inverter in figure 4.1a was run through the performance optimizer with several loads on the output. An example of one of the resulting gates is shown in figure 4.1b. A graph of the transistor width versus load is shown in figure 4.2. As expected, it is linear.

Optimizing a single inverter can have dramatic effects on the delay of the output signal. Figure 4.3 is a plot of load versus delay for an unoptimized inverter as estimated by the simple RC model in Andy (solid line) and as measured at the $V sub EQ$ point in Spice (+). The Andy curve was scaled to superimpose it on the Spice graph. Both are linear and both show the problem with heavily unbalanced loads. The × marks are the Spice simulation results for the optimized inverter. The delays are approximately at the four transistor load delay, a result of setting the fanout factor to four.

4.1.2. A Shift Register Cell

The shift register shown in figure 4.4 was run with the same loads as the inverter in the previous example. The graph of the width of the pulldown transistor (upper line) and the pass transistor (lower line) in the cell are shown plotted against load in figure 4.5. A major point of interest on the plot is the load above which the pass transistor width changes. The larger pass transistor makes a larger load on the output node for the inverter, so those transistors must must be made larger. This leads to the slight upward bend in the pullup transistor width line at the point where the pass transistor size starts changing.

Simulation results for the shift register are shown in figure 4.6. The Andy

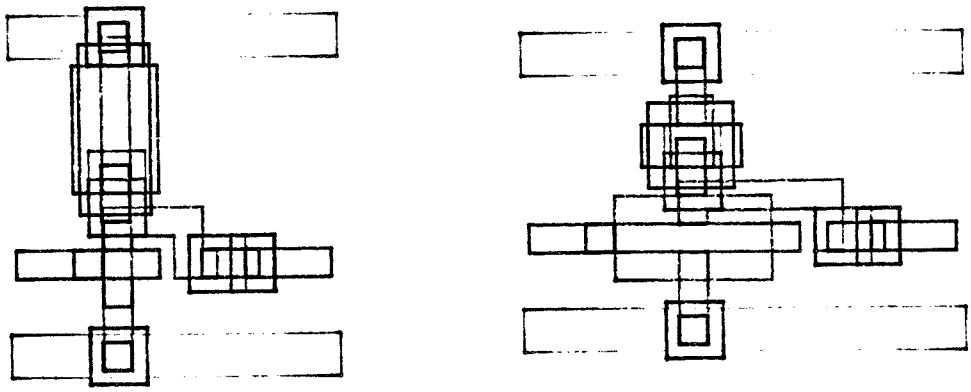


Figure 4.1. An Optimized Inverter. a) One Transistor Load on Output. b) Twenty Transistor Loads on Output.

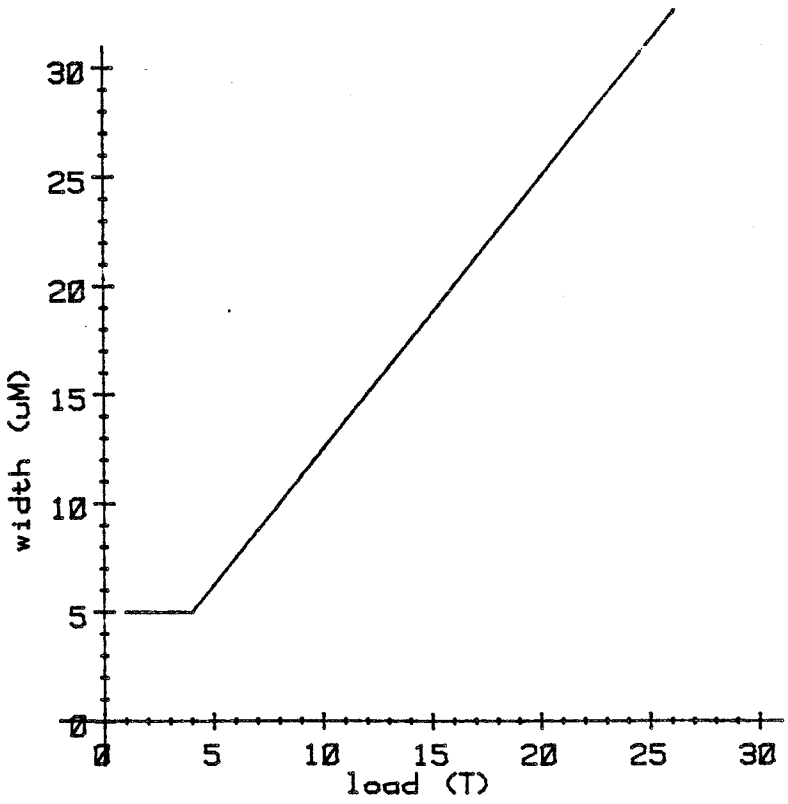


Figure 4.2. Plot of Transistor Width Versus Transistor Loads for the Inverter Cell in Figure 4.1.

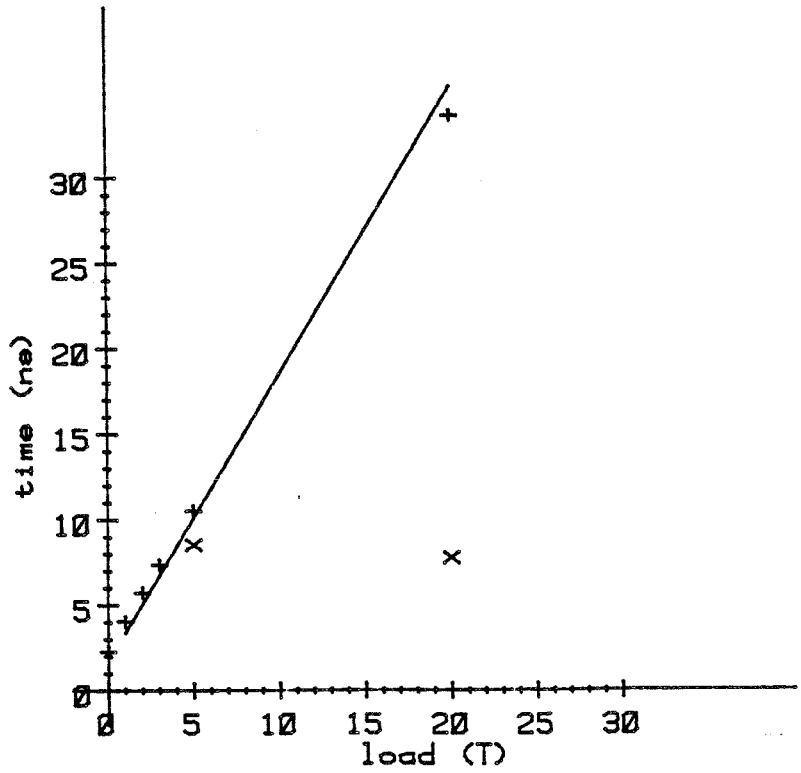


Figure 4.3. Plot of Output Delay Versus Transistor Loads for Inverter Cell

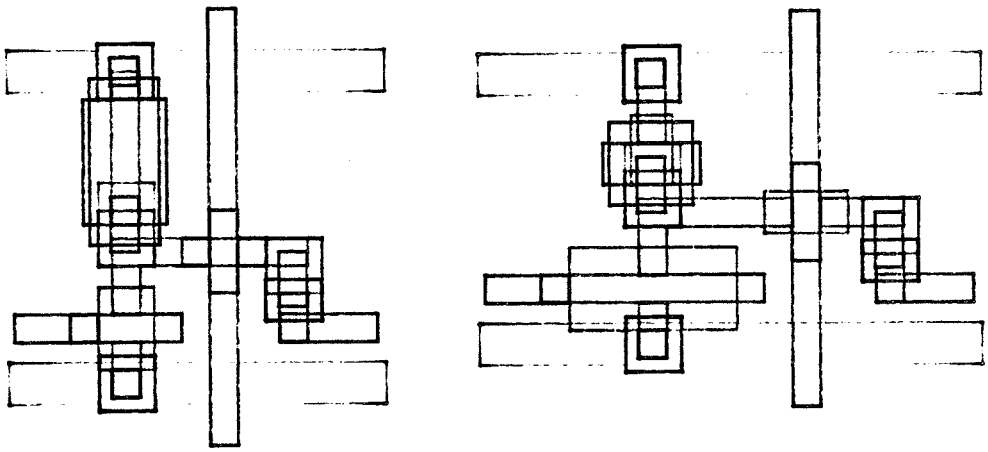


Figure 4.4. A Shift Register Cell. a) One Transistor Load on Output. b) Twenty Transistor Loads on Output.

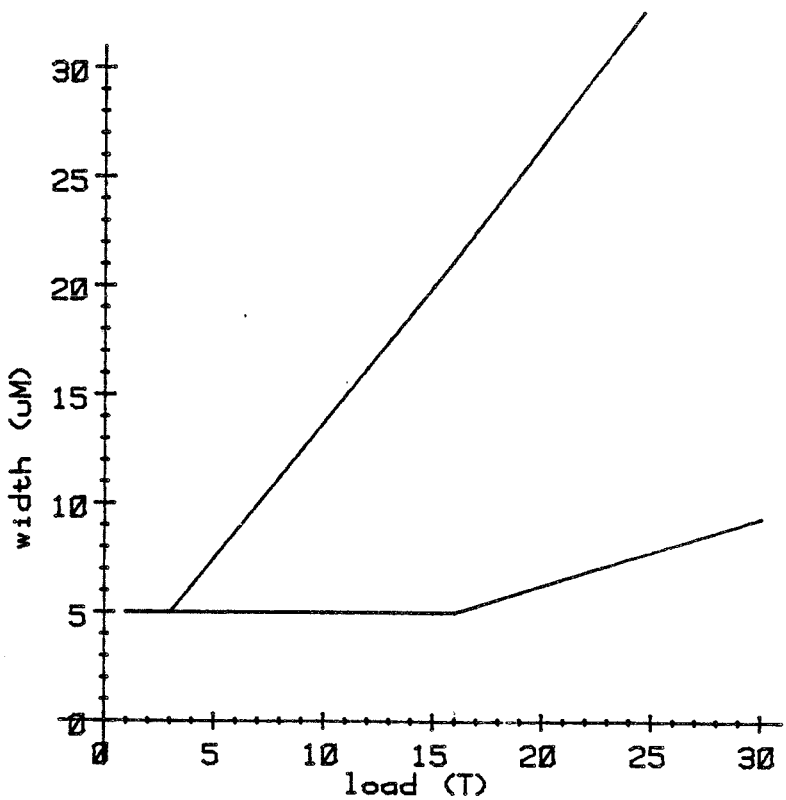


Figure 4.5. Plot of Transistor Width Versus Transistor Loads for the Shift Register Cell in Figure 4.4.

delay estimate is shown as the solid line, Spice simulation results as "+" for unoptimized delays and "×" for optimized delays. Again, optimizing the devices yields great performance advantages. Notice also for the twenty transistor data point, the relatively minor delay penalty from not sizing the pass transistor (the asterisk).

Although the increase in speed for sizing the pass transistor as well is minor, it comes with very little extra cost in power and area. The power cost is due to the larger driver needed to balance the load of the larger pass gate. Area costs are low because pass transistors are usually placed between restoring logic stages that constrain the size of the cell.

An examination of typical integrated circuits shows that there are few cases where pass transistors feed large loads. The most common cases are tri-state output pads and bus structures. Adequate treatment of bus structures may become more important in the future with larger and more complex chips. It will be very important to guarantee some reasonable decisions on the bus drivers, so pass transistor sizing may become critical in the future.

4.2. A Chain of Gates

Figure 4.7 shows a short chain of gates. This chain of gates was put through the performance optimizer with a variety of loads on the output. The purpose was to show the ramped scaleup to drive the load. This ramp can be seen in the graphs in figure 4.8 for several values of the capacitive load. The load is measured in number of minimum transistor loads.

The table in figure 4.9 compares delays measured from Spice simulation for

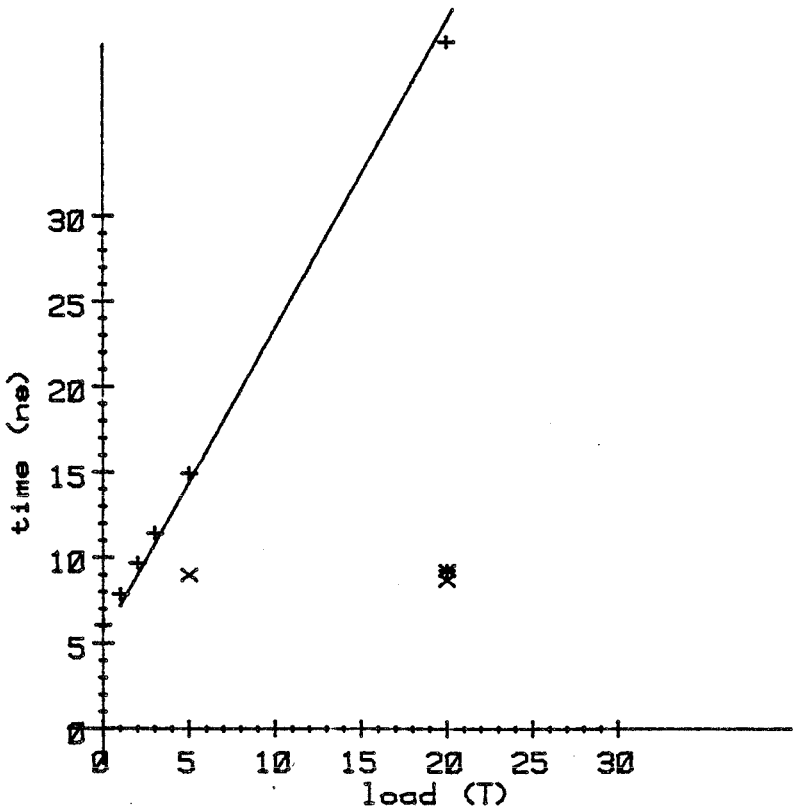


Figure 4.6. Plot of Output Delay Versus Transistor Load for Shift Register Cell

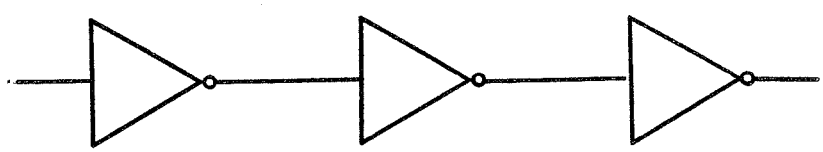


Figure 4.7. A Chain of Gates

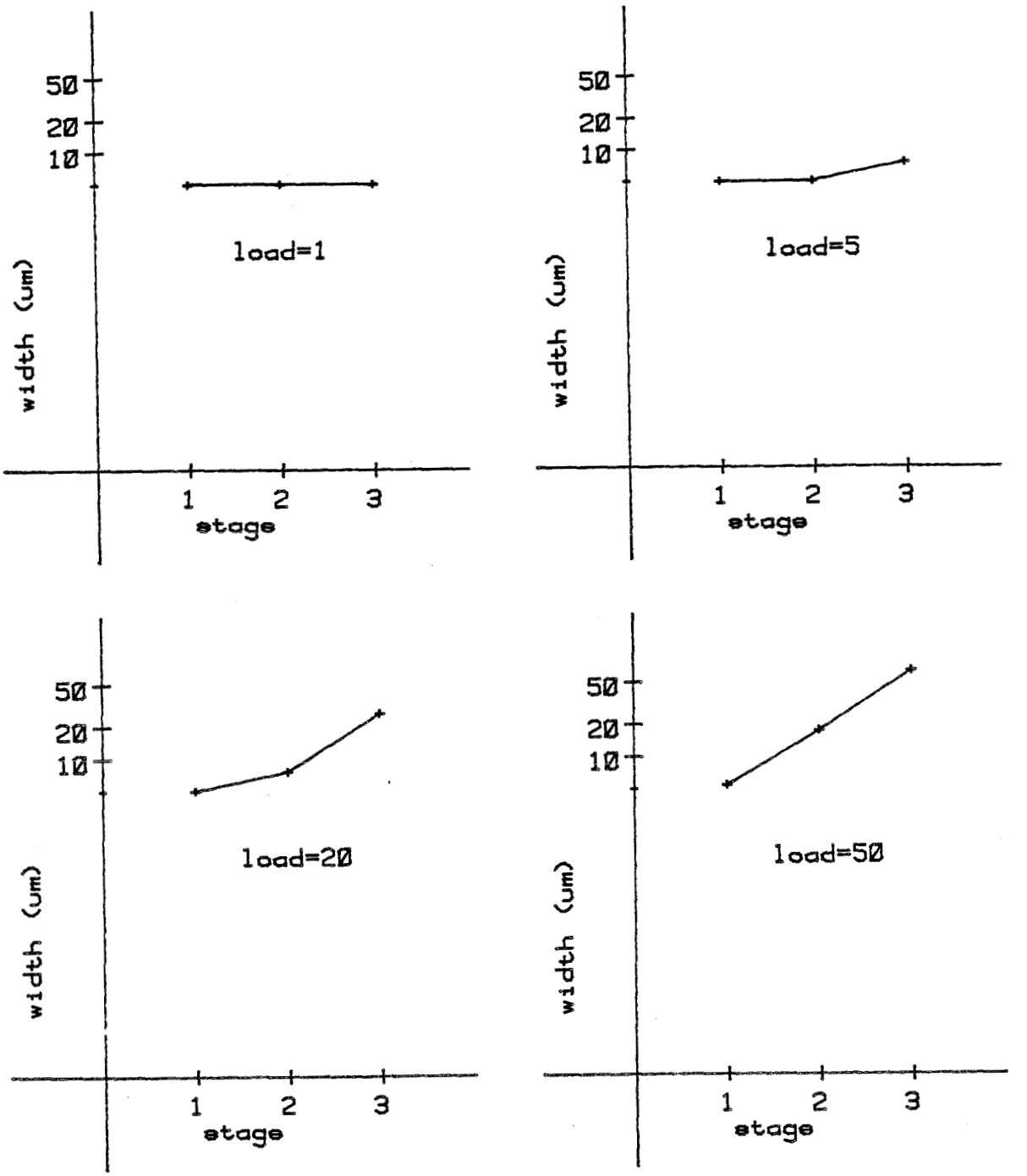


Figure 4.8. Plots of Gate Stage Versus Transistor Widths for Several Capacitive Loads.

various output loading. The first set of numbers are for no load, the second set for larger load and the final set for a relatively large load. Qualitatively, the results are not astounding -- larger transistors makes faster gates. But quantitatively, the results are surprising -- a great deal of additional performance can be squeezed from common designs.

4.3. Power Optimization Examples

These examples show the small-scale effects of power optimization on a few simple circuits. The savings can be important in larger circuits. A summary of the results is shown in the table in figure 4.10. Notice that in both cases, power optimization improves performance. This is the result of decreasing the load on a minimum-sized transistor, as described in chapter 2.

	delay (ns)	power(μW)	area(λ^2)
No Load			
Unoptimized	15.91	114	1080
Optimized	15.91	114	1080
Optimal Delay	15.91	114	1080
Load = 5T			
Unoptimized	23.56	114	1080
Optimized	20.19	142	1093
Optimal Delay	13.76	275	1080
Load = 20T			
Unoptimized	46.54	114	1080
Optimized	21.35	359	1255
Optimal Delay	14.61	656	1404

Figure 4.9. Statistics for a Chain of Gates.

	del	pwr	dxp
Unrelated Paths			
Original	1.12	1.00	1.12
After Delay Optimization	.423	3.14	1.33
After Delay and Power Optimization	.422	2.75	1.16
Fanout Example			
Original	1.16	1.00	1.16
After Delay Optimization	.533	3.22	1.72
After Delay and Power Optimization	.498	2.67	1.33

Figure 4.10. Power Optimization Results

4.3.1. Unrelated Paths Example

The simplest case of power optimization occurs when two unrelated paths are present in a cell, as is the case in figure 4.11. One path is the upper path from the input on the left through the one inverter to the output. The other path is from the lower left input through the three gates along the bottom and out the connector on the right.

The upper path is loaded with a 15 transistor load and the lower path with a twenty-five transistor load. These loads could be reasonably expected in parasitics, since a twenty transistor load is produced by a a 200λ polysilicon run, about the length of 10 half shift register stages. The delay-optimized and power-optimized versions of the cell are shown in figure 4.11. Notice that the transistors in the gates along the non-critical path are made smaller by the power optimization.

4.3.2. Fanout Example

Figure 4.12a shows a simple circuit with fanout. Both paths are loaded as described in the unrelated paths example. The lower path is the critical path, since there are more gate stages in it. The cell after delay

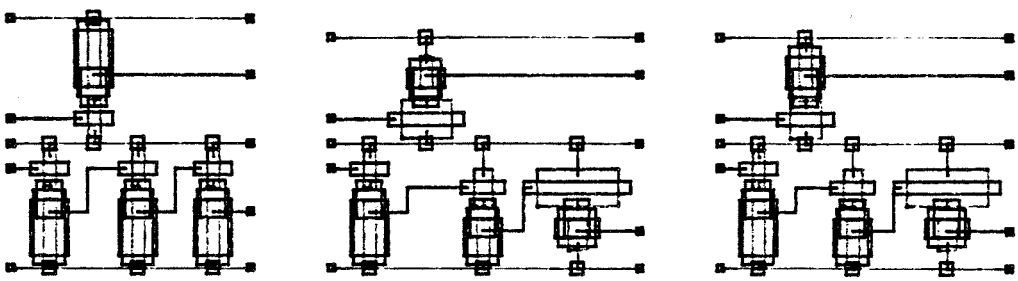


Figure 4.11. Unrelated Paths Example. a) Original.
b) After Delay Optimization. c) After Power Optimization

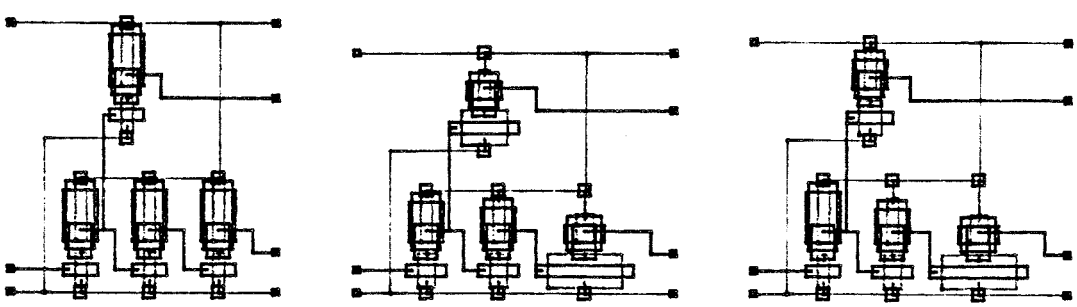


Figure 4.12. Fanout Example. a) Original.
b) After Delay Optimization. c) After Power Optimization.

optimization is shown in figure 4.12b. All gates were made larger to drive the load optimally. However, the transistor on the upper path need not be that large. When the upper transistor is made smaller, the transistor that fans out to the other gates can be made smaller also.

The power optimization slows down the upper path so its delay is the same as the delay along the critical path. The transistors in the upper path gate are made smaller, so the capacitance on the output of the fanout gate is less so that gate is made smaller also, as can be seen in figure 4.12c.

4.4. Larger Examples

The two designs in this section are respectably large since they represent significant parts of a design and involve some reasonably complex interactions of gates and nodes. These examples are included to give an understanding of how Andy works on a real chip and to show the kinds of improvements Andy can make.

4.4.1. The Logical Filter Example

The *logical filter* chip calculates the Boolean sum of products on a stream of input bits, given a set of constants. It was designed and fabricated to test the capabilities of Riot [Trimberger 1982b], a simple graphical chip assembly tool. It predates Andy, so it makes an impartial, if not state-of-the-art test case. In addition, it is a sample of a machine-composed chip, so it allows us to evaluate Andy in the composition environment in which it will most likely be used.

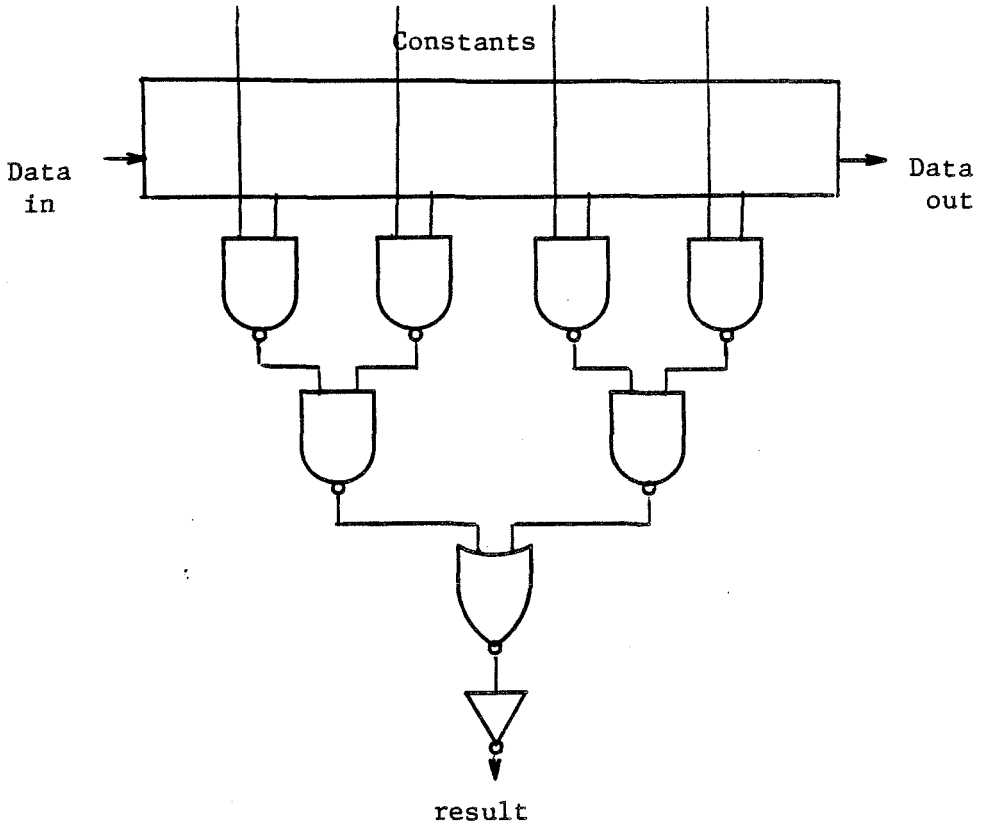


Figure 4.13. Logical Filter Chip Gate Diagram.

A schematic gate-level diagram of the logical filter chip is shown in figure 4.13. It contains a few dozen transistors and some respectably long interconnection runs. The bonding pads could not be used in Andy because they were defined geometrically. The part of the chip used as an example is shown outlined in figure 4.14.

The table in figure 4.15 compares the performance of the logical filter chip before optimization, after delay optimization and after delay and power optimization. The example was run with the parasitic capacitance both on and off. The same runs without parasitics are shown in the second part of

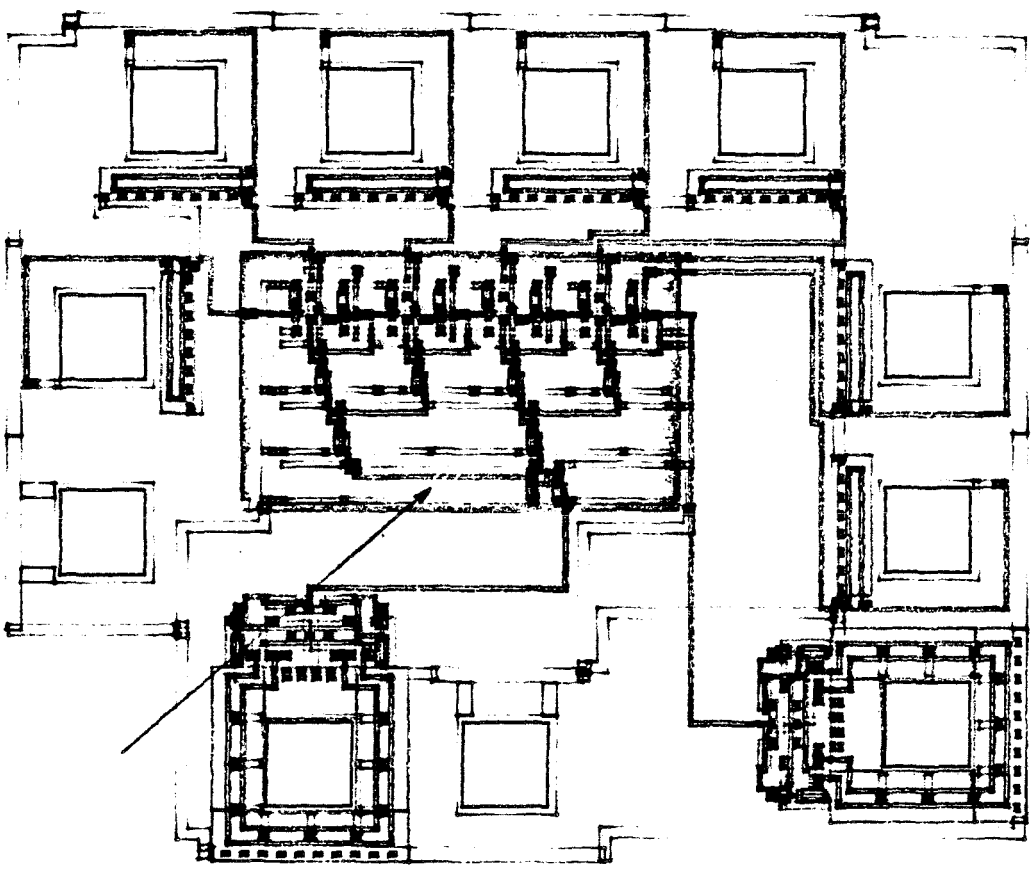


Figure 4.14. Logical Filter Example.

	del	pwr	d×p
With Parasitic Capacitance			
Original	3.22	4.00	12.9
After Delay Optimization	2.00	9.55	19.1
After Delay and Power Optimization	2.00	7.74	15.5
Without Parasitic Capacitance			
Original	1.32	4.00	5.28
After Delay Optimization	1.46	3.44	5.02
After Delay and Power Optimization	1.46	3.44	5.02

Figure 4.15. Table of Logical Filter Results.

the table.

The performance optimizer was able to cut the delay for the signal to be ready by about 40 percent. The delay-power product was not as good as the original, but not unreasonable, either. The power optimization was not very effective because there are only a few gates off the critical path.

The numbers for the case without consideration of parasitic capacitances after delay and power optimization were the same because all devices were made minimum size during delay optimization and couldn't be optimized any further for power. The delay numbers are a little worse because some scale-up was put into the original circuit and was eliminated by the optimizer as described in chapter 2.

The transistor sizes without the parasitic capacitance are nearly identical to the original hand-optimized circuit. This is to be expected, since the rules for sizing gates typically refer to the number of gates and not the parasitics. This designer did not concern himself with parasitic capacitance. The resulting delays are calculated assuming no parasitics and imply that the optimization is pretty good. However, if the circuit is optimized as if the parasitics are insignificant, then delays measured including the parasitics,

the results do not seem as good.

One point of particular interest is the sizing of the NOR gate at the bottom of the circuit (arrow in figure 4.14). When parasitics are ignored, the transistors are made minimum size. When parasitics are included, they are much larger than the minimum. At first glance, it would appear that the former is correct, since the gate must only drive one transistor, the pulldown of the inverter. However, closer inspection shows that the parasitic capacitive load is very large on that node, because the NOR gate was stretched quite a distance by the assembly tool. The diffusion line that connects the two pulldown gates (and is part of the output node for the NOR gate) was made very long and its capacitance amounts to several gate capacitances, requiring larger transistors to drive it.

Finally, this example was run with a number of different desired fanout factors. The results of this run are shown in the table in figure 4.16. As expected, delays shrink and power consumption rises considerably with smaller fanout factors. Also of note is the delay-power product which improves dramatically with larger fanout factors.

Fanout Factor	del	pwr	dxp
2.000	1.04	112	116
2.718	1.39	21.3	29.6
4.000	2.00	7.74	15.5
8.000	3.31	3.59	11.9

Figure 4.16. Table of Logical Filter Results with Different Fanout Factors.

4.4.2. PLA Example

The Programmable Logic Array (PLA) shown in figure 4.17 is a Sticks version of the traffic light controller example in [Mead 1980]. Currently, PLAs are laid out on a regular grid with all transistors the same size. However, each gate in the PLA, a horizontal slice in the AND-plane or a vertical slice in the OR-plane, drives a different load, depending on the number of transistors and the size of the transistors on it. In addition, large PLAs have large parasitics associated with the wires. Finally, the outputs on the PLA may be required to drive large loads. All these effects cause PLAs to be wasteful of speed on the heavily loaded paths and wasteful of power on the lightly loaded paths.

In the traffic light controller, a twenty transistor load was placed on the *Start Timer* output (ST in figure 4.17) to simulate a long conductor to a timer. The design was then passed through the delay optimizer and power optimizer. The resulting layout is shown in figure 4.18. Notice that the array is still regular, because all transistors in a gate are sized the same. This leads to whole columns in the OR-plane and whole rows in the AND-plane being sized identically. But different gates are sized differently. So, while the array is still a rectangular grid, the grid spacing is increased or decreased where the gate sizes were changed. One can follow the effects of the increased load on the *Start Timer* output. The output transistor is larger, so the transistors in the gate that make up the OR-plane gate for that output are wider, causing the OR-plane to be wider in that column. Wider gates in the OR-plane column lead to larger transistors in the AND-plane rows that drive them. These in turn lead to larger drivers.

Another point of interest is the size of the two input drivers on the right side

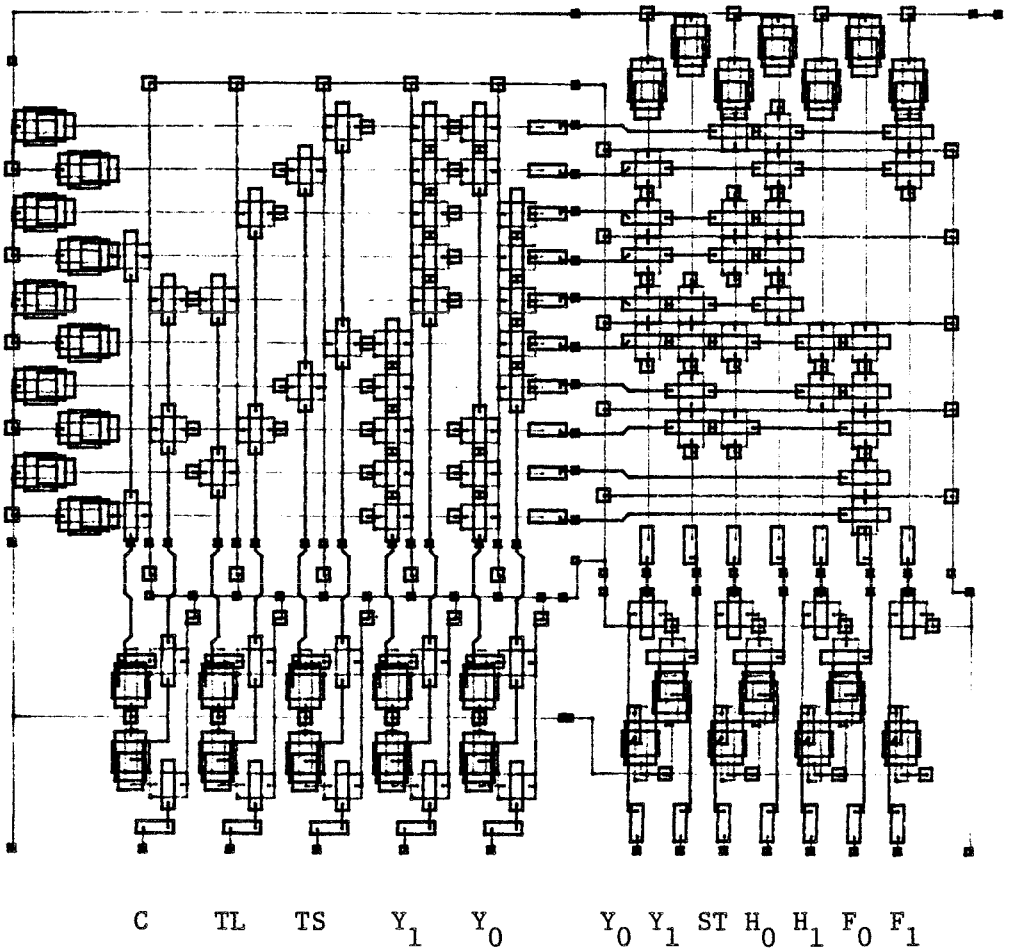


Figure 4.17. Programmable Logic Array Example

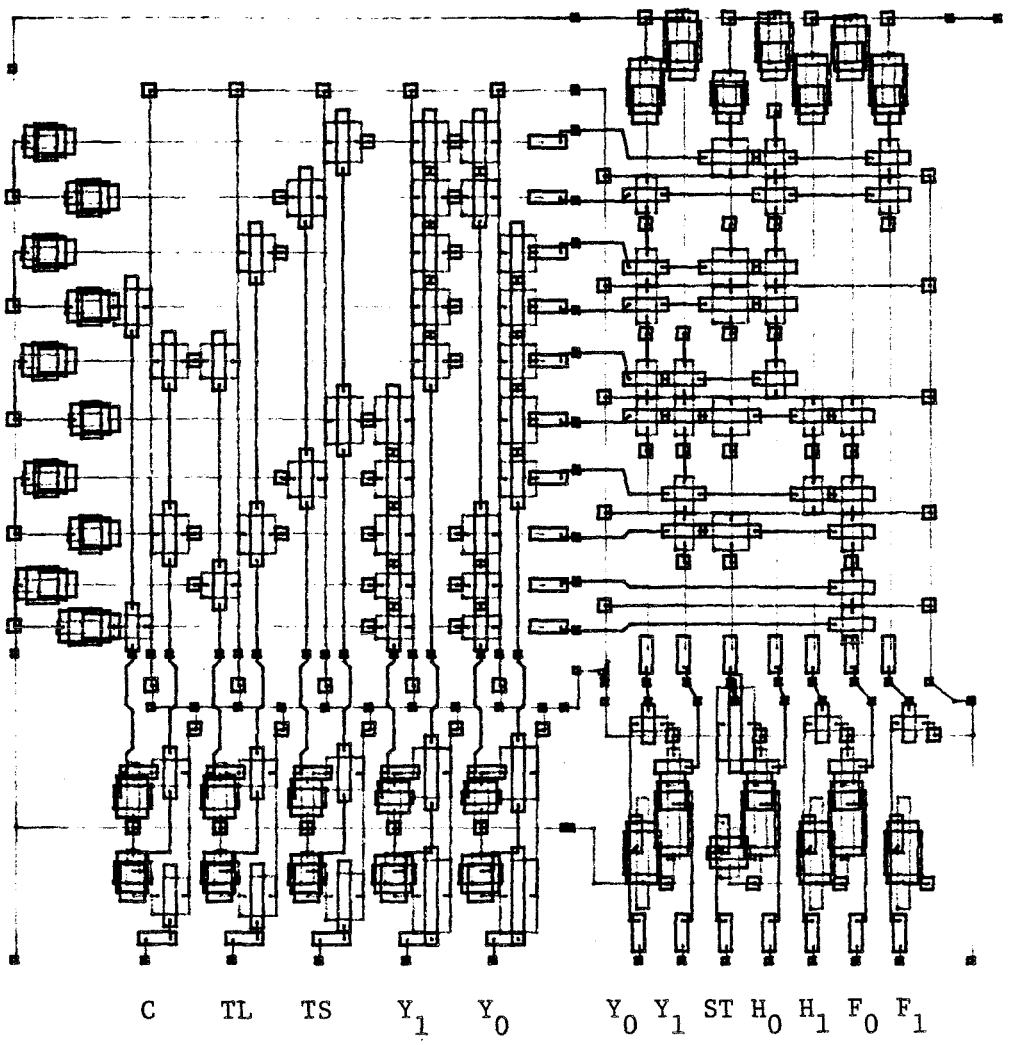


Figure 4.18. Optimized Traffic Light Controller PLA.

of the PLA inputs. These are the drivers for the feedback terms, and they must be made large because of the number of minterms they drive. They were vastly undersized in the original layout.

The table in figure 4.19 compares the unoptimized and optimized versions of the PLA. The numbers are all unscaled estimates from Andy. PLAs prepared in this fashion can still be made without human intervention. Fast logic need not be difficult to produce.

4.5. Summary of Examples

The Andy optimizations improve performance by approximately forty percent in larger designs, and improve the delay-power product as well. Power consumption is increased, as is area. While the area increase is rather small. the power penalty of approximately twenty percent may be unacceptable to some.

The use of the Andy performance optimizer to improve the performance of automatically-generated designs, such as the PLA, and machine-composed designs like the logical filter, shows the need for such a tool. The assembly tools sometimes cause additional delay problems for a designer by creating parasitics as part of the connection mechanism or by overloading a single

	delay	power	d×p	area
Unoptimized	1.63	17.0	27.6	22518
Optimized	.964	22.4	21.6	24705

Figure 4.19. Statistics for the Traffic Light Controller PLA

node in the PLA. Because of the automation of the assembly tools, the designer cannot take these problems into account when the design is specified. They must be addressed after the connection has been specified. Andy is essential in these cases to avoid costly design iterations.

CHAPTER 5

The Andy Performance Optimization Algorithms

This chapter describes the algorithms used in Andy in more detail than they were covered in chapter 3. These algorithms are concerned with node generation, gate recognition, performance optimization, and power optimization. Possible variations on these algorithms are discussed in the following chapter.

5.1. Overview of the Algorithms

The Andy performance optimization algorithm attempts to optimize fanout between gates in the circuit, attempting to make the ratio from one gate to the next as close as possible to the values that best meet the performance design rules specified in chapter 3.

The algorithm is broken down into pieces in figure 5.1. The input to Andy is in Sticks form. Sticks twigs are merged into electrical node segments inside the cells and the node segments from all instances in the hierarchy are merged into full electrical nodes. Gates are then derived from the nodes and transistor structure.

After the gate and node data structure has been constructed, the optimization algorithm proceeds backward through the net of gates, sizing every gate for which the load on the output node is known. Known loads are parasitic interconnection loads, gate capacitances of transistors that have been sized, and outputs from the circuit. Feedback paths are broken by

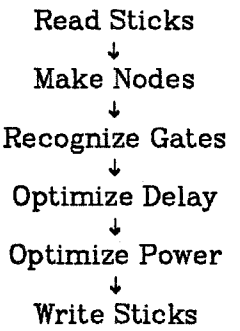


Figure 5.1. Performance Optimization Flowchart.

simply choosing a gate and sizing it. When a transistor size is changed, it causes the gate that drives it to be re-sized, since the load on the output of that gate has changed. Therefore, feedback paths will be cut, then sized properly.

The final optimization step is the power optimization step in which the critical delay path is determined and all paths off the critical path are slowed down to match the critical path delay. This saves power in those places where high speed does not improve the overall performance of the cell.

5.2. Finding Nodes

Electrical nodes must be extracted from the Sticks representation before gates can be recognized in the circuit. An electrical node consists of all twigs and component connector references in an equipotential region [Sutherland 1979]. The node extraction is fairly easy, since each of the components in the Sticks form has a simple electrical definition.

The node determination for a cell is done in three parts. First, all the node segments in the cell are found. These node segments consist of a Sticks twig,

all the component connector references on the twig, and recursively includes other twigs and component connector references on electrically equivalent connectors on the components. Node determination scans through contacts and electrically common connection locations on transistors and connectors.

Node segment determination is done for all cells that have instances in the cell in which we are doing the node determination. These node segments are collected in the cell, then merged into complete electrical nodes. The merge algorithm crawls up and down the design hierarchy coalescing node segments across cell boundaries. Finally, as the nodes are merged, the components in the node are separated into drivers of the node and loads on the node.

Formally,

Def. A *connection* is a pair (t, r) where t is a Sticks component and r is a Sticks connector name.

Def. Two connections (t_1, r_1) and (t_2, r_2) are *equal* if $t_1 = t_2$ and r_1 and r_2 are electrically equivalent. The person who defines the atomic Sticks components is responsible for stating which connectors are electrically equivalent (See the Sticks Standard Definition).

Def. Two Sticks twigs are *directly connected* if they include equal connections.

Def. Two twigs t_a and t_b are *connected* if there is some sequence of twigs T such that T_i and T_{i+1} are directly connected for all i, and t_a and t_b are in T.

Def. A *node segment* is the largest set of connected twigs inside a cell and all

connections on those twigs.

Def. A *node* is the largest set of connected twigs and all connections on those twigs inside a cell including all twigs inside instances in the cell.

5.2.1. Node Segment Determination

Node segment determination in a cell starts at the Sticks twigs. The twig is added to the node, and all components connected to the twig are scanned as outlined in the pseudo-code below. A reference to the specific connector on a component is included in the node and all other twigs that refer to electrically equivalent connectors on the component are added also. In the case of a contact component, all connectors are electrically equivalent, so all other twigs that refer to the contact are included in the node. References may be made to connectors on instances, twigs, contacts, and transistors. The algorithm proceeds recursively until nothing more can be included in the node, then a new twig is chosen to start a new node. The node segment determination for a simple cell is shown in figure 5.2.

```
PROCEDURE include_twig(tw);
IF twig not in a node already THEN BEGIN
    include tw in node;
    FOR all components in this twig DO BEGIN
        include component reference in the node;
        FOR all twigs connected to electrically equivalent connections on
            components DO include_twig;
    END;
END;
```

At the time node segments are made in the instances included in the cell, a pointer is placed in the instance that points back to the parent instance. This pointer makes the design hierarchy doubly linked so the *scanloads algorithm*, which is described below, can scan both up and down through the

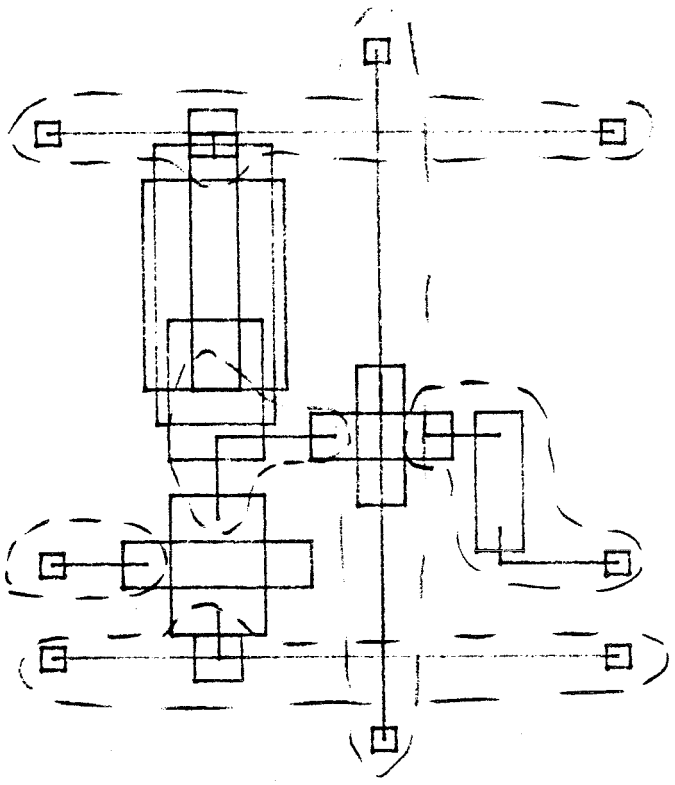


Figure 5.2. Node Segment Derivation from a Simple Cell

hierarchy. The scanloads algorithm also requires that every connector in all cells have a reference to its node segment in the cell. This pointer is also added to connectors during the node segment determination. The resulting data structure is shown somewhat schematically in figure 5.3.

5.2.2. Merging Node Segments

When all twigs are included in node segments, all the node segments from instances are brought into the cell for merging into complete electrical nodes. The merge crawls up and down and across the design hierarchy to

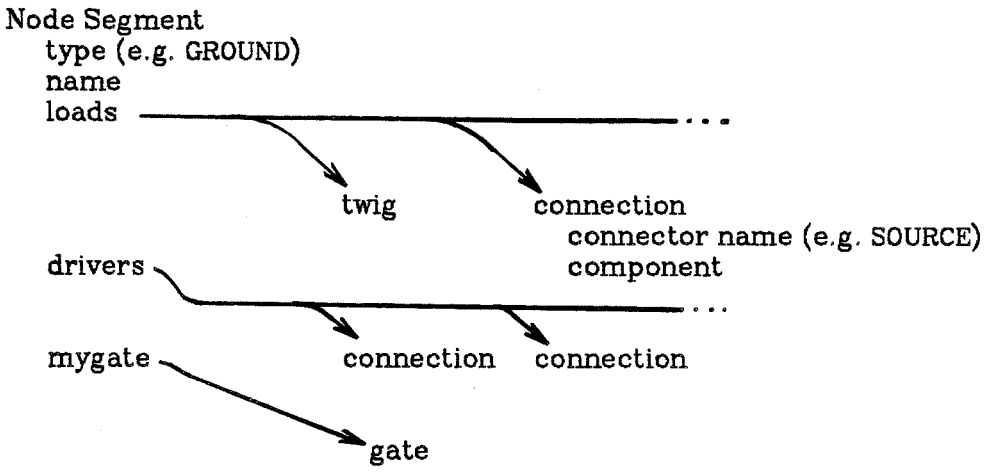


Figure 5.3. The Data Structure for Node Segments.

determine full electrical nodes from the node segments. Merge uses a recursive algorithm called the *scanloads algorithm*, shown in figure 5.4.

The scanloads algorithm follows a node anywhere in the hierarchy and may

```
PROCEDURE scanloads(nod,proc,inst);
REF(node) nod;  PROCEDURE proc;  REF(instance) inst;
IF NOT nod.scanning THEN BEGIN
    PROCEDURE xSL(th);  REF(thing) th;
    INSPECT th
    WHEN instance  DO
    scanloads(THIS instance.findconnector(myconn),proc,THIS instance)
    WHEN connector DO
    IF inst.parent=/=NONE
        THEN scanloads(myinst.node(this connector),proc,inst.myparent)
        ELSE proc(th)
    OTHERWISE proc(th);

    nod.scanning := TRUE;
    nod.loads.apply(xSL);
    nod.scanning := FALSE;
END of scanloads;
```

Figure 5.4. The Scanloads Algorithm.

come back up the hierarchy to touch other nodes in the parent cell. The parameters are the node to scan, a procedure to be invoked on each element in the node, and the instance in which this node resides. The instance is necessary for following the design hierarchy upward. When an instance is encountered, the algorithm recursively calls *scanloads* on the corresponding node inside the instance. When it reaches a connector, it finds the node on the containing instance that connects to the current instance, and invokes *scanloads* recursively on that node. If there is no containing instance, then the current instance represents the cell in which we are finding nodes so the connector must be kept. All other things in the node: twigs, contacts, transistors and resistors, as well as the connectors on the top-level instance are passed to the procedure.

Scanloads uses a flag in the node to keep cycles in the node description from causing the algorithm to loop forever. Another improvement not shown here for clarity sake is a fourth parameter, the "exception". The "exception " is a component that the algorithm will skip. Since some path must have been followed to get to the current node, it is not necessary to check the component reference through which that path arrived. The exception parameter stops the algorithm from looking back at the node on the other side of the instance or connector it just followed to get to the current node.

Merge starts by invoking a scanloads procedure on an unused node segment in its list with the copy procedure. An *unused* node segment is one that has not yet been merged into a complete node. The copy procedure simply copies the component it is given into the new merged node from the node segment. Therefore, all twigs, contacts, transistors, resistors and top-level connectors are copied into the merged node.

As the contents of each node segment are copied into the merged node, the node segment is removed from the list of unused node segments. Also, constraint information, node type and node name, if applicable, are copied into the merged node. For clarity in the algorithm above, these operations are not shown. Figure 5.5 shows the node segments and the final node determination for a shift register piece. Notice that a node may contain twigs and components from anywhere in the hierarchy.

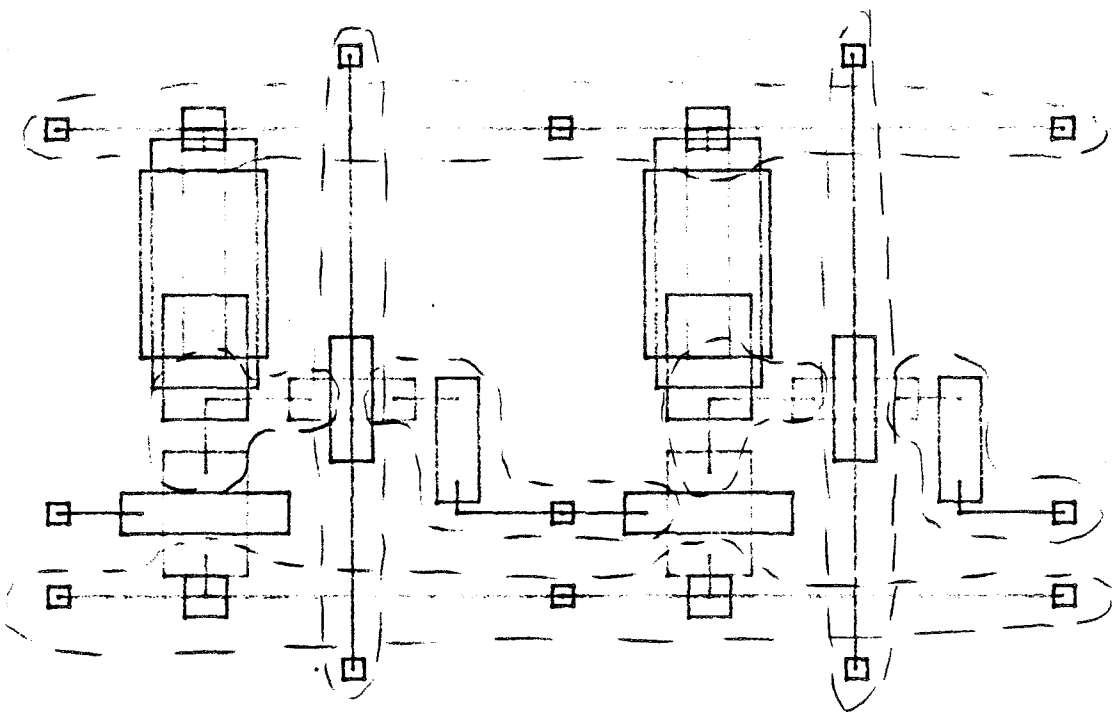


Figure 5.5. Nodes in a Shift Register Segment

5.2.3. Segregation of Drivers and Loads

As the node merge copy procedure adds components and twigs to the merged node segment, it separates them into *drivers* of the node and *loads* on the node. The discrimination is necessary for the gate finding algorithm and improves the speed of the optimization algorithm. Twigs and contacts are always loads on a node. Transistor gates are loads, but transistor source and drain are drivers. Depletion pullup transistors are drivers, but the gate connection is a load.

Connectors outside the cell are added as drivers or loads or both, depending on the type of the connector. INPUT, IO, POWER, GROUND, BUS and CLOCK connectors are drivers of the node. OUTPUT, IO, BUS, and CLOCK connectors are loads. It is possible for a connector to be both a driver and a load on a node. This is the case with source and drain connections on pass transistors also, which are discovered and handled later in the gate finding algorithm.

The copy procedure also gives every component in the node a pointer back to the node that drives it. This pointer is necessary for later operations, such as critical path determination. Transistors keep pointers to all three nodes: the gate, source and drain. The source and drain nodes are needed in the gate finding algorithm, and all three are used during performance optimization.

Nodes inherit types from the connectors on them, if any. POWER and GROUND connectors are of particular importance because POWER and GROUND are not driven by any of the transistors on them, they supply the drive for the transistors. The final pass over the nodes moves all transistor source and drain references on POWER and GROUND nodes from drivers of the nodes to

loads on those nodes. BUS connection types are also propagated to the node. BUS types are used to terminate the gate finding step which is described in the next section.

5.3. Finding Gates

The gate network is derived from the node representation of the cell which includes the complete electrical nodes derived from the entire hierarchy and the Sticks transistors, resistors, contacts and twigs. Gate determination from this form is possible, if it is assumed that the circuit contains only *well-formed* gates and that there are no serious logical flaws in the circuit.

The gate finding step will be successful if the design is composed of only well-formed gates. Well-formed gates are those with a single pullup transistor or resistor and a tree-like pulldown connected to GROUND. There may be pass transistors, but there may not be gates with multiple pullups. Gates with general graph-like pulldown structures are not allowed, either. Figure 5.6 shows two examples of well-formed gates.

Formally,

Def. A *restoring logic gate* is a triple (u, d, o) where u is a transistor called the "pullup", d is a tree of transistors called the "pulldown structure" whose leaves are connected to GROUND and whose root is connected to the output node, and o is a node called the "output node".

Def. A *transmission gate* is a transistor that is not along a path from POWER to GROUND or a transistor that is constrained to be a pass transistor.

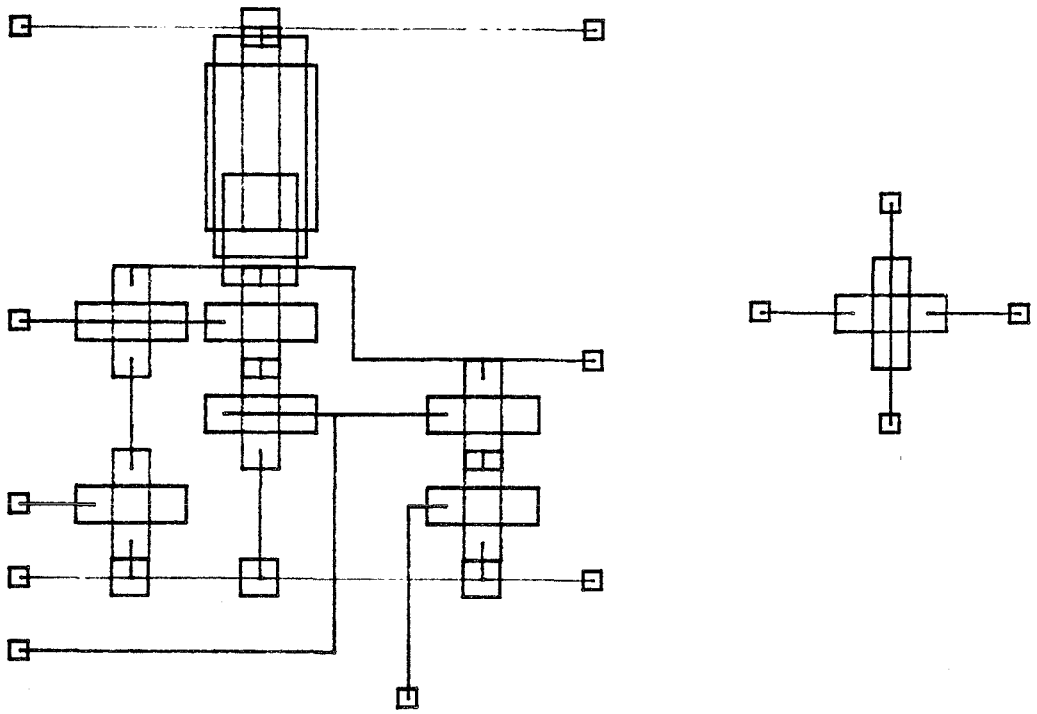


Figure 5.6. Well-Formed Gates

Def. A *well-formed gate* is a restoring logic gate or a transmission gate.

During the construction of the data structure, Andy checks the circuit for serious flaws in the network. POWER shorted to GROUND is detected and flagged in the node finding step. POWER and GROUND separated by a single transistor is caught along with other ill-formed gates in the gate recognition step. Ill-formed gates that do not involve shorting of POWER and GROUND are caught in gate recognition when a single transistor is found to belong in two different gates or at two different places in the same gate. Since intelligent resolution is not possible, the construct is flagged as an ill-formed gate.

As shown in the pseudo-code below, the gate finding algorithm finds gates by following the POWER node to a transistor source or drain. Since one side of the transistor is connected to POWER, it must be a pullup for a restoring logic gate, so a new gate is created with the transistor as its pullup. Although, in the usual case, the transistor is a depletion mode device used as a load resistor, other forms for super-buffer gates and precharged gates are legal as well.

```
PROCEDURE find_gates;
FOR all POWER nodes DO BEGIN
    FOR all transistors on the node DO BEGIN
        make a new gate.
            the pullup is the transistor.
            the output node is the node opposite the POWER
        FOR all paths of transistor source and drain from the output node DO
            IF the path leads to GROUND
                THEN make them pulldowns of the gate
                ELSE make them transmission gates
    END;
END;
```

The node on the other side of the transistor is the node that the gate is driving, which must be the output node of the gate. The gate finding algorithm follows that node to find the pulldown transistor structure. When a connection to the source or drain of a transistor is found, there are two possible situations: the other side of the transistor may or may not connect to GROUND. If the other side of the transistor does not connect to GROUND, the transistor is remembered and the node on the other side of the transistor is scanned recursively, building a tree-like structure pointing to the transistors. The recursion stops when the GROUND node is found or if there are no source or drain connections on the node.

If the node is the GROUND node, then all the transistors on the path from the gate's output to GROUND must form a NAND network, serial connection to

GROUND in the gate. Parallel connections to GROUND make NOR-type connections. If there is no GROUND connection, the transistors along the path must be pass transistors, and a new transmission gate is made for each pass transistor. Figure 5.7 shows the gate determination of a simple example.

The gate finding algorithm can differentiate between pulldown transistors and pass transistors in most situations. However, some circuits, such as the shared bus in figure 5.8, confuse it. The algorithm sees a path from both pullups through the BUS node, the pass transistors and the pulldown on the

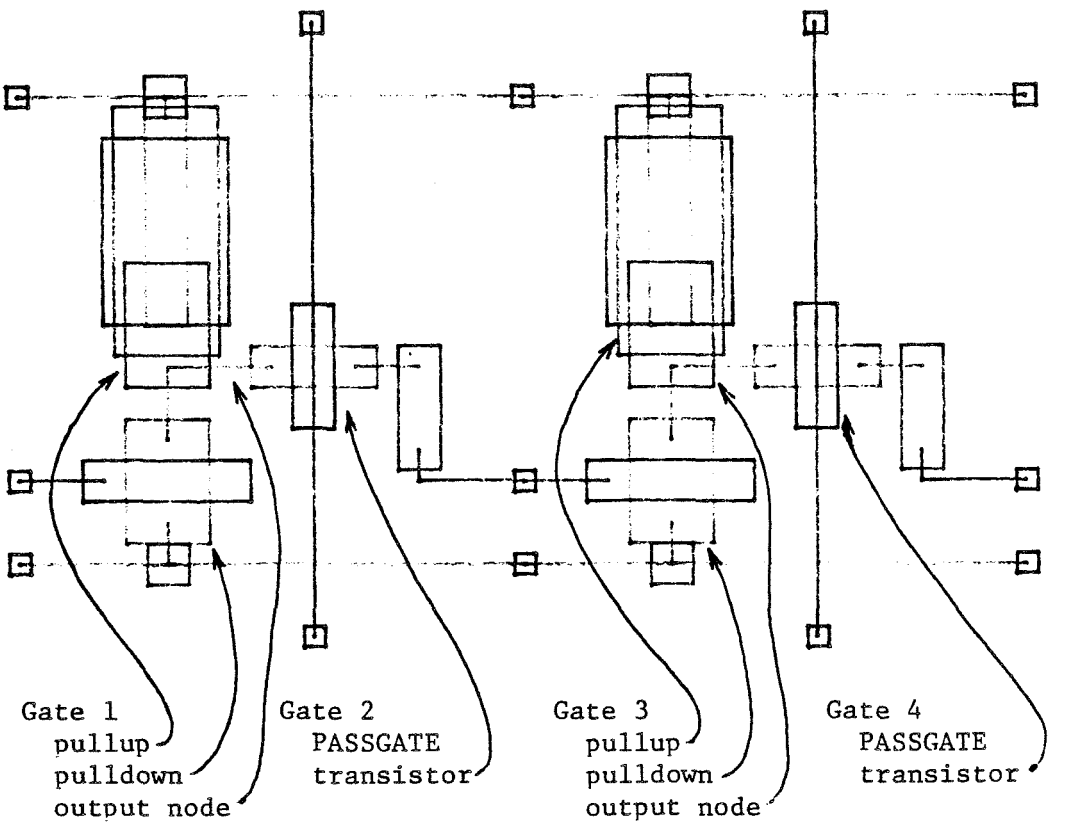


Figure 5.7. Gate Determination of Shift Register

other bus driver to GROUND. This improper interpretation can be avoided by explicitly declaring the pass transistors or by constraining the bus node.

If a transistor has been constrained to be a pass transistor, the recursion stops, the gate determination ends, and the transistor is made into a transmission gate. If a node is found of type BUS, then the gate finding algorithm is similarly terminated. These constraints help remove confusion in some MOS structures that do not fall into the category of well-formed gates described above, but which occur frequently in designs. These structures include some more exotic transmission gate logic as well as the shared bus described above.

As gates are made, the output node of a gate is given a pointer back to the gate that drives it. This pointer is required later in the delay optimization step. Figure 5.9 gives a schematic view of the node and gate data structure that the gate determination completes for use in the delay optimization step.

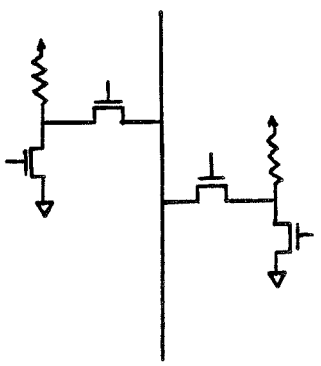


Figure 5.8. Shared Bus Structure

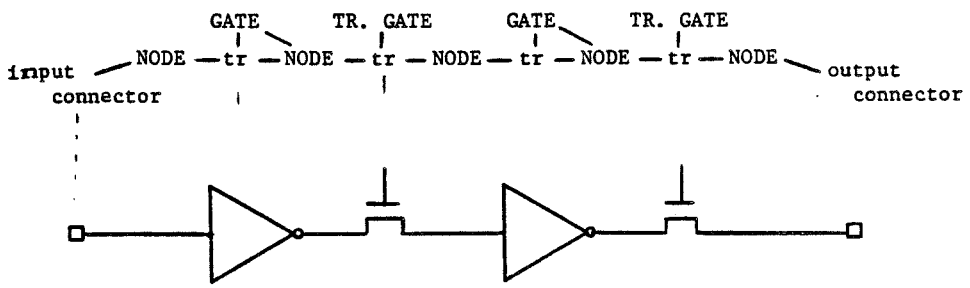


Figure 5.9. Node and Gate Data Structure.

5.4. Performance Optimization of Gates

The sizes of the transistors in a gate are determined by the resistance required for the pullup and pulldown structures to charge or discharge the capacitive load on the node in a reasonable time. "Reasonable time" is defined by a user-set-able parameter that represents the number of minimum sized transistor capacitances that can be charged or discharged by a minimum sized enhancement pulldown transistor resistance in that "reasonable time". This number is called the *fanout factor* because it is the ratio of the gate transistor size to the load size. A discussion of the fanout factor and its effect on the performance of a circuit is included in chapter 2.

I now define some terms that are used throughout this section,

Def. Two nodes are *possibly connected* if they are separated by a chain of transmission gates.

Def. The *load* on a node N is

1) if the node contains a twig with a constrained load, then the load is the constrained load, otherwise
2) the sum of the capacitances of all twigs in all possibly connected nodes to N and all capacitances of all components in those nodes.

Four definitions that will be useful in the following section are also given here:

Def. A node has a *known load* if the node contains a twig that has a constrained load or if all the transistors in all possibly connected nodes are in gates that were sized.

Def. A *ready gate* is a gate whose output has a known load.

Def. The *delay* of a gate is the length of the pullup divided by the width of the pullup times the load on the output node.

Def. The *power consumption* of a gate is the width of the pullup divided by the length of the pullup.

5.4.1. Gate-Oriented Performance Optimization

The performance optimizer first sets all constraints in the circuit, including device size constraints and loading constraints on connectors. Then it iterates, finding all gates for which all loads on the output node have been determined, and sizing them. No gate can be sized until all transistors on its output node have been sized. External connectors on the cell being optimized have a minimum load or a constrained load, so the gate sizing starts at the outputs and works backward toward the inputs.

The iteration continues while there are gates yet to be sized. If a pass through the gates yields no gates that can be sized but there are still some unsized gates, then a feedback must exist in the gate structure. The smallest case where this occurs in functional circuits is the cross-coupled NAND latch in figure 5.10. As shown in figure 5.11, the algorithm picks one of the gates and sizes it, breaking the feedback. The transistors on the gate are now defined loads, so the other gates in the chain can be sized, also. When a transistor is sized, it marks the gate that drives it as "unsized", because its load has changed. So that gate goes through the sizing algorithm again. Thus, the sizing will proceed around a feedback loop, eventually returning to the gate chosen to break the loop. All gates in the loop will eventually be run through the resizing process with the correct loads on

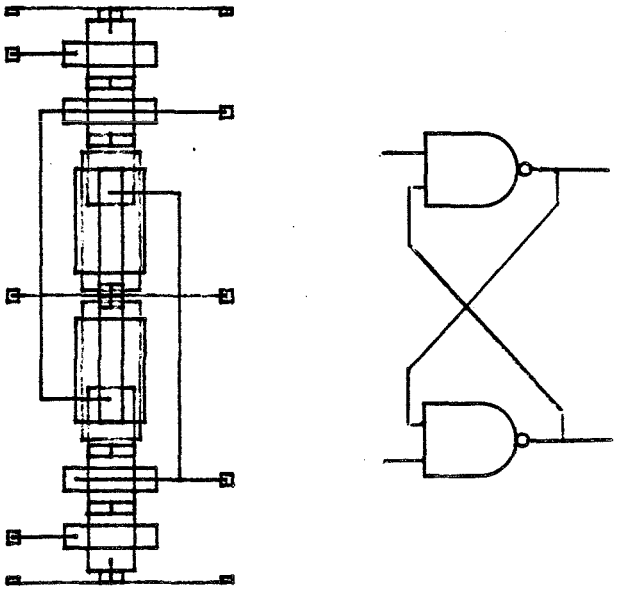


Figure 5.10. Cross-Coupled Gates.

their outputs. The gate that was sized to break the feedback will be re-sized after the last gate in the feedback loop is processed.

To keep the chain of re-sizing gates from continuing forever, transistors only cause their driving gates to be resized if the transistor changed size significantly (by more than 5%). It is easy to prove termination of this algorithm: at each stage the change in a gate is a constant factor less than the gate before it. Even a gate that has its output connected to its input sees geometrically decreasing changes in the resizing it must do.

A gate is sized by first finding the capacitive load on its output. If a node has a constrained capacitance, then that capacitance is used, otherwise the load determination totals the gate capacitances from all transistor gates on the node and, optionally, the parasitic capacitances of the wires that make up the node.

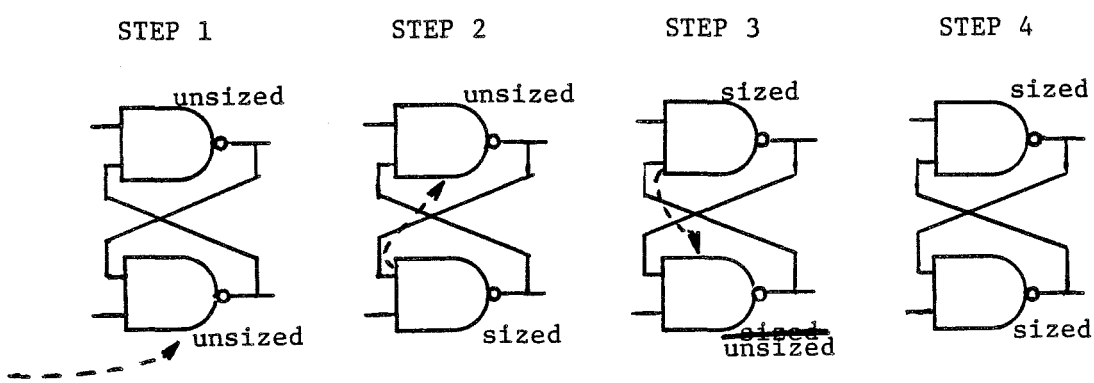


Figure 5.11. Sizing Gates in a Feedback Loop.

The load determination looks through pass transistors pessimistically, assuming the gate will have to drive all loads on the far side of all pass transistors simultaneously. This pessimism is visible in figure 5.12, in which both gates will be sized to drive C_1+C_2, even though logical analysis reveals driving the separating pass transistor, P, may require that G_2 only drive C_2. Analysis of the logic for further optimization is beyond the scope of this work.

5.4.2. Sizing Transistors in Restoring Logic Gates

The capacitance is divided by the *fanout factor* and converted to a desired transistor resistance, expressed as a desired width to length ratio of the pulldowns, using a conversion factor derived from the ratio of the minimum sized transistor capacitance and the per-square transistor resistance for a minimum sized transistor. If the pullup transistor gate is not connected to the output of the gate, the desired resistance is cut to one quarter, accounting for increased performance of super buffer and precharge gates.

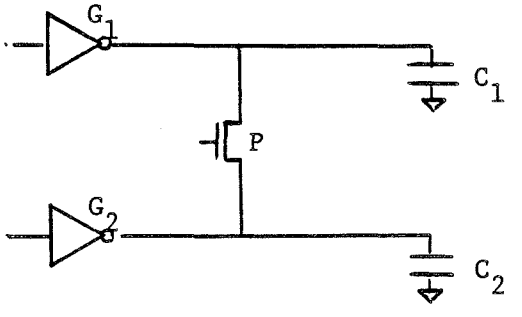


Figure 5.12. Load Calculation Looks Through Pass Transistors Pessimistically

Enhancement-mode pullups, such as precharge transistors are treated slightly differently. The resistance of resistor-like pullups is set to four times the resistance of the pulldown, as demanded by the design rules in chapter 3. The resistance of the precharge transistor, however, is made the same as that of the corresponding pulldown transistor. This is not desirable in all situations, however, so in precharging applications where the precharging device is not time critical, the designer may wish to constrain its size.

The pullup/pulldown ratio is preserved in NAND structures by increasing the pullup resistance in proportion to the number of serial transistors in the longest pulldown chain to GROUND.

Finally, the pulldown transistor size is set to the width/length ratio that gives the proper resistance for the transistor, as calculated from the capacitance of the output node, the number of gates in the longest NAND chain in the gate, the type of pullup device and the kind of signal on the gate of the transistor.

The resulting equations for the width to length ratio for pullup and pulldown transistors are:

$$Z_{pu} = \frac{C_L}{f_{tr} f_{ff} k_{all} k_{pu}}$$

$$Z_{pd} = \frac{C_L N_{pd} f_{ps}}{f_{ff} k_{all}}$$

where C_L is the load capacitance on the output node, N_{pd} is the number of serial gates in the NAND chain in the gate pulldown, f_{tr} is the factor which is 1 for resistor-like pullups and 4 for transistor-like pullups, f_{ff} is the fanout factor, the number of minimum transistor loads to be driven by a minimum

transistor driver, k_{all} is a constant that includes the constants to convert capacitance in units of picofarads to a desired resistance and from that to a transistor width. k_{pu} is the basic ratio between an nMOS depletion-mode pullup transistor and an nMOS pulldown transistor, f_{ps} is the pass transistor factor which is 1 if the signal on the gate of the pulldown transistor is a restored logic signal and 2 if it has passed through a pass transistor.

When a pulldown transistor is sized, if the signal that drives the transistor gate is gated by a pass transistor, the pulldown transistor is made twice as wide to compensate for the lower gate voltage.

5.4.3. Sizing Transistors in Transmission Gates

Because of their bidirectionality, transmission gates cannot be sized until the loads on the nodes on both sides of the pass transistor have been defined. The size of the pass transistor is set so that the resistance of the pass transistor is the same as the resistance of the pullup on an inverter driving the the larger capacitance on either side of the pass transistor. This keeps the pass transistor from becoming a serious impediment to the speed of the circuit, while avoiding unnecessarily large pass transistors. The pass transistor ratio is sized using the same equations as the pulldown, with some simplifications because there is no NAND chain and the signal on the gate of the pass transistor may not be gated by a pass transistor (that condition is flagged as an error).

$$Z_{ps} = \frac{MAX(C_{L_1}, C_{L_2})}{f_{ff} k_{all}}$$

5.4.4. Transmission Gate Chains

Frequently, transmission gates occur in chains, as in the case of the Manchester carry chain. Such a chain is shown schematically in figure 5.13. This section describes how the techniques already presented optimize such a chain.

The gate selection algorithm finds that none of the gates are ready to be sized, since the inverter size depends on the sizes of the transmission gates and the transmission gates themselves depend on the sizes of the other transmission gates. Therefore, one of the gates is chosen to break the loop. The algorithm will continue to pick one of the transmission gates to size until all but one are sized. Then the last transmission gate will be sized since all its loads are known. The other transmission gates may have to be re-sized, and this process continues until all the gates reach mutually acceptable sizes, within the five percent cutoff.

In the end, the last transmission gate in the chain, C, will be sized to drive the larger load, presumably C_L. Transmission gate B will see that its larger load is $C+C_L$, so it will be slightly larger. Transmission gate A will be sized to

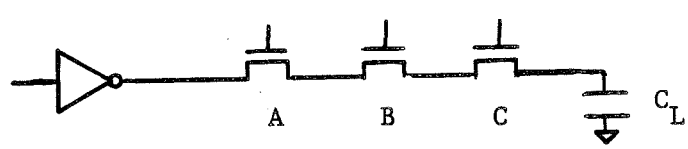


Figure 5.13. A Chain of Transmission Gates Driven by an Inverter

drive $B+C+C_L$. The inverter will see all pass transistors plus the load capacitance as its load and will be sized accordingly. The result is a linear increase in the sizes of the transmission gates.

5.5. Power Optimization Off the Critical Path

Fast gates off the critical path do not contribute to the overall speed of the circuit, but they do consume more power and use more area than slow gates. Therefore, the final step of performance optimization is concerned with lengthening short delays in order to reduce power consumption in parts of the circuit where delay is not critical.

This power optimization is done in two parts. First, the gate network is analyzed and gates that are off the critical path are marked with desired delays that will allow them to run more slowly without making any path delay longer than the critical path delay. Afterward, the device sizing algorithm from the delay optimization operation is run to set device sizes to match the desired delays in each gate.

The node and gate data structure shown in figure 5.9 can be viewed as a

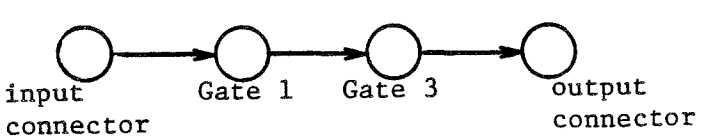


Figure 5.14. The Directed Graph Corresponding to the Circuit in Figure 5.7

weighted directed graph with restoring logic gates as the vertices of the graph and the electrical nodes that are the outputs of the gates as the edges of the graph. Each arc is weighted by the delay for the gate to drive the capacitance on the node (figure 5.14). Transmission gates are referenced by pointers on the electrical nodes, and are not nodes in this graph because they do not contribute to the power usage. Loops in the graph would result from feedback structures in the circuit. A more complex graph (figure 5.15) results from the fanout example from chapter 4.

Power optimization is carried out on paths of gates through the circuit. A *path*, as used in this section, is a chain of gates in which all gates are distinct. The representation of the path in the directed graph is thus the same as the definition of *path* given in [Harary 1972]. A path can be identified by the final electrical node in the path, which is the last node to be driven. That node may be on an output connector on the cell or it may be on the gate of a transistor in a gate in the cell.

We are concerned with *critical paths*, the longest path between any two points, as measured by the sum of the weights on the arcs connecting the points (the sum of the delays).

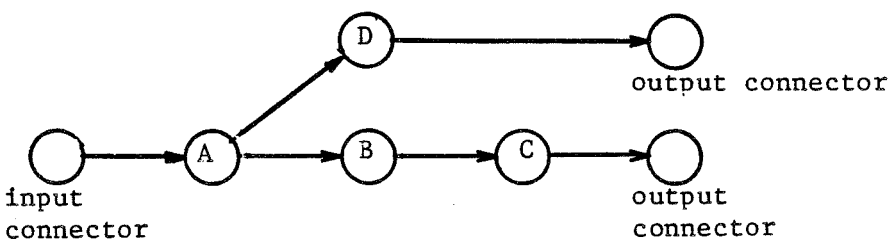


Figure 5.15. The Directed Graph Corresponding to the Fanout Example

Therefore, for the power optimization, we make the set of the longest paths in the directed graph that start at the points that correspond to inputs of the cell and end at points that correspond to the cell's output connectors or to inputs to gates. Shorter path segments form separate paths, but paths that are subsets of longer paths are not included.

The resulting paths include all gates in the cell, and every gate will belong to a chain along its most time critical path. When clocks are recognized as breaking the paths, paths may start and end at pass transistors that are gated by CLOCK signals.

Formally,

Def. The *power optimization* graph, $G(V,E)$ of a circuit with node set S_n and gate set S_g is a weighted directed graph such that

1) The vertices include
    1) restoring logic gates,
    2) connectors, and
    3) if the CLOCKing option is on (see below), then include two new vertices for each pass transistor.
2) The edges are
    1) (i, j, d) if i is an input connector, where d is the delay due to either a minimum-sized transistor driving the load or a transistor smaller than the load by the fanout factor, whichever is smaller, as discussed in the following section,
    2) (i, j, d) if i and j are restoring logic gates and j includes transistors in its pulldown transistor tree that are on a node that is possibly connected to the output node of i, where d is the delay of gate i,

3) (i, j, d) if i is a pass transistor, where d is as described in 1),

4) $(i, j, 0)$ if j is a pass transistor, and

5) $(i, j, 0)$ if j is an output connector.

Def. An *i-j critical path*, π, on the power optimization graph G is a path in G such that for all paths between vertices i and j, π has maximum weight.

Def. The *critical path* of a power optimization graph is the maximum weight i-j critical path.

5.5.1. Path Determination

The path determination scans first the nodes on output connectors on the cell, then the nodes on the transistors in gates. The node is followed backward to either an input node or a transmission gate that is driven by a node of type CLOCK. All paths starting at a CLOCK node are checked as part of the critical path also. The use of the CLOCK as a path delimiter is optional and can be turned off.

The path determination does a depth-first backward search of the graph shown in figure 5.15, following the node connected to each of the pulldown transistors on every gate it encounters. The maximum of the delays to a gate is saved in the gate as is the delay to drive its output node (that delay is the product of the gate's pullup length/width and the output node capacitance).

In this delay calculation, all input nodes are assumed to be driven by a gate that would have been produced by the performance optimization algorithm. That is, every input is assumed to be driven by a gate that is either scaled

down by the fanout factor from the capacitance on the input node, or minimum-sized, whichever is greater. These delays are used later when optimizing paths inside the cell. The path determination from figure 5.15 is shown in figure 5.16. Notice that a gate may be part of more than one path. This is corrected in further processing.

Paths that start at a gate that is a member of a longer path are known as *fanout* paths. Paths that end at a gate are called *fanin* paths. There are two other kinds of paths, those that are unrelated to other paths, *unrelated*

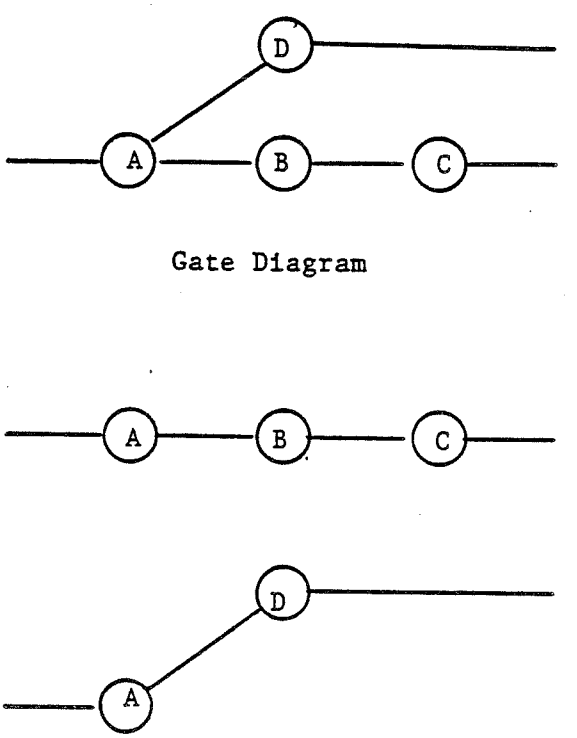


Figure 5.16. The Paths Determination from the Graph in Figure 5.15

paths, and those that are both fanout and fanin paths, *fanboth* paths. All four kinds of paths are shown in figure 5.17.

The path determination algorithm described above does not recognize fan out, since it follows the path all the way to the input connector, and lengths of paths are not known until after all paths have been found. Fanout and fanboth paths are uncovered on the second pass through the gates when the gate delays are actually set. So the paths seen by the delay adjustment are those in figure 5.18. When all paths have been found, they are sorted in order of decreasing delay so paths with longer delays are sized before those with shorter delays. The longest delay path is the critical path for the cell.

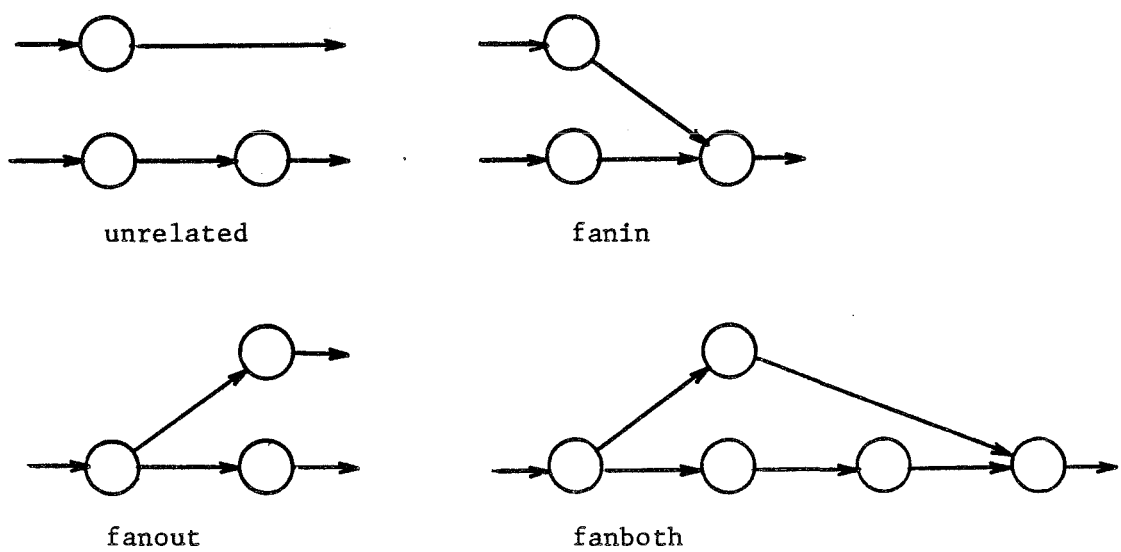


Figure 5.17. The Four Kinds of Paths.

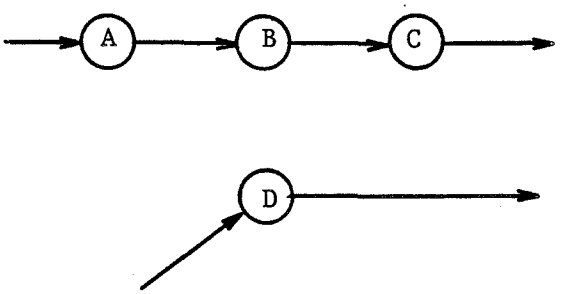


Figure 5.18. Paths Seen by the Delay Adjustment for the Path Determination in Figure 5.16.

5.5.2. Path-Oriented Power Optimization

Power optimization attempts to make all paths through a cell take as much time as the critical path does. Gates in unrelated paths must be adjusted so the delay of the path is the same as the critical path. Gates in fanout paths must be set so the path length is the same as the delay from the gate at the start of the path to the total delay for the cell. Gates in fanin paths must be set so the delay of the path is the same as the critical delay to the gate at the end of the path. Fanboth paths must be lengthened so the delay is the same as the delay between the gates at the ends of the path.

Instead of making the all paths in the cell as long as the critical path, the user may set a desired delay for the entire cell. If that delay is longer than the critical path, it is taken as the total delay for unrelated paths and there is no critical path in the cell, so the longest path goes through the power optimization just like other paths.

When the paths are found, each path is given a pointer to the gate that

terminated it. The delay to that gate, which might change during the power optimization, is the final time for the path. The starting time for the path may be zero, if the path leads to an input on the cell. If some gates in the path have already been set by the power optimization, the starting time is taken as the delay from the last gate that has already gone through the power optimization. When this is done, the path is shortened before optimization, cutting off the already-optimized gates. This shortened path is either a fanout path or a fanboth path, as mentioned above.

The path delay must be set to the difference between the final time and the starting time. There are several options as to how to divide this extra delay among the gates in the path. If we only care about matching the delay, we could simply choose one gate and make it much slower to use all the extra delay. Alternatively, we could make all gates have the same delay, dividing the desired delay evenly.

We could divide the extra delay evenly among all gates in the chain, making each gate slower than it was by the same amount of time. Or, we could make each gate slower by the ratio of the desired delay to the current path delay. It is not clear what the optimum solution is, and the solution used in Andy is the last one: each gate in the chain is made slower by the ratio of the desired delay to the current path delay, so each gate shares in the power saving.

The gate sizing algorithm in the performance optimizer uses the desired delay field on the gates to set the size of the devices if the desired delay is present. The power optimization simply sets those desired delays then runs the performance optimizer. The performance optimization algorithm sets device sizes to match the desired delays, which are set to zero for delay

optimization.

Transmission gates are not explicitly modified by the power optimization algorithm, since pass transistors do not affect the power consumption. Pass transistor sizes may change, however, as a result of the power optimization if the capacitances on its nodes are reduced.

CHAPTER 6

Performance Optimization Options

This chapter deals with the myriad ways that Andy could have been. Some of these alternate approaches were tried and found to be lacking. Some were thrown out without being tried. Some are simply other ways of doing performance optimization that may be as valid as the one implemented in Andy.

There are also several suggestions for future work in this area, centering around improved algorithms and more accurate delay models.

6.1. Explicit Parametrization of Delay, Power and Area

It has already been stated that long before physical limitations limit the speed of a circuit, area use and power requirements become ridiculous. A system that optimizes performance cannot do so without regard to other design parameters. Rather than use an algorithm that has the side effect of limited delay, as I have done, one could envision a system that allows the designer to specify the relative importance of the various design constraints such as power, speed, and area, and let the system choose a design that fits them all.

Such systems are called *multiple criterion optimization* systems [Lightner 1981]. These systems typically rely on *heuristics*, techniques for design that are derived from a designer's experience, to lay out the circuit and trade off

one design criterion against another. Unfortunately, the search for an optimum in the multiple criterion decision space can be very time consuming, so these systems have been used only on very small circuits. Improved heuristics may be available in the future that will enable such systems to produce reasonable chips in a reasonable amount of time.

Multiple criterion optimization systems also require a way to express the relative importance of various design criteria. However the desired optimizations for an integrated circuit are pretty well determined by the time the circuit is in a form suitable for machine optimization. To be specific, one wants optimum delay along the delay critical path an optimum density along the dimensional critical paths. In all places where there is slack space and delay, one wants low power. Although a system like the one described in this thesis could provide that function, a more versatile system would give the designer even more power to trade off design constraints very late in the design cycle.

6.2. True Optimum Delay

As discussed in chapter two, the Andy performance optimization algorithm does not make gate chains with true optimal delay. The optimal delay requires a constant factor scale up through the entire chain of gates. It should be possible to optimize small chains of gates using this uniform ramping thereby achieving optimum delay along the chain.

The correct constant scaling factor for minimum delay is relatively simple to calculate in simple structures like a chain of inverters, but rather difficult to calculate in more complex structures. Attempts to address this problem in a

gate array system [Ruehli 1977] led to a rather complex solution, since the size of every gate is dependent on the size of all gates in all chains that intersect the chain of which the gate is a part.

True optimum was abandoned in Andy for three reasons. First, because the algorithms were too complex for a system that was even remotely interactive. Such time consuming algorithms may be made faster, or new computer hardware may make them more attractive in the future, causing a re-evaluation of this reason. The second reason for avoiding true optimum of all gates is because the designers do not now design chips where every gate is optimally fast. Designers are not concerned with delay to the exclusion of all else. A casual survey showed that circuits produced by human designers have most geometry at minimum size and only a few gates larger to make the circuit faster. These designs usually have good area and power statistics.

The third and most important reason for abandoning true optimum is the quality of the results obtained with the heuristic. Since the heuristic produces circuits that are within about twenty five percent, not much more improvement could be expected with a more accurate algorithm.

6.3. Transistor-Oriented Performance Optimization

An early attempt at the delay optimization algorithm used a transistor-based optimization method. This was assumed to agree with MOS circuits that may include a considerable amount of pass transistor logic.

A simple transistor sizing algorithm takes each transistor independently and sizes it according to the load it drives. There are two difficulties with this

approach. First, MOS transistors typically are bidirectional, and it is impossible to tell which side is the load. This problem can be solved by marking the POWER and GROUND nodes, and assuming that no devices drive POWER and GROUND nodes and that there are no transistors that will make a short circuit between POWER and GROUND. Therefore, in the inverter in figure 6.1, the transistors drive the the output rather than POWER and GROUND nodes. These are the same assumptions made in the current system during the gate extraction step.

The second problem with this method is more severe and is shown in figure 6.2: the size of the pullup transistor on the NAND gate depends on the number

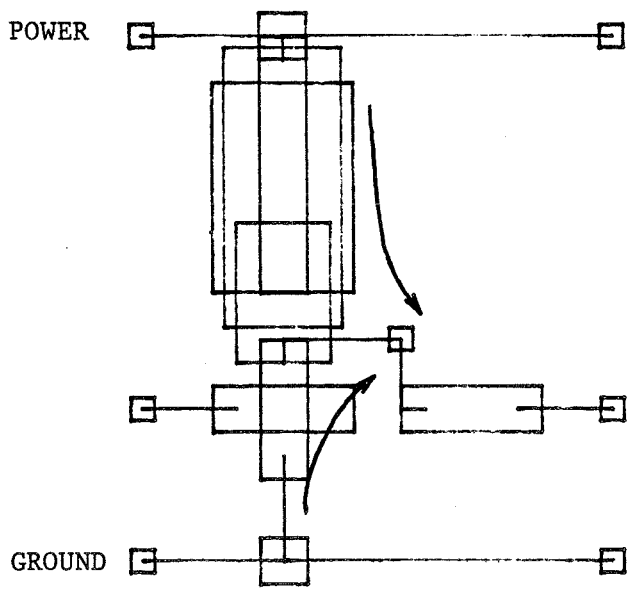


Figure 6.1. An Inverter

pulldown transistors in the longest NAND structure in the gate. If the gate structure is not known, the pullup cannot be sized correctly. This could be solved by requiring the designer to provide proper pullup/pulldown ratio. However, the structure of the design may not allow the designer to know how many transistors are in series in the pulldown.

In order to size NAND structures properly, then, the system must have a description of the circuit in terms of gates. This causes some problems with MOS circuits, since a traditional gate-like description is frequently an unacceptable description. Because so much MOS logic is made with transmission gates, transmission gates were added as a separate case to Andy's repertoire of gates.

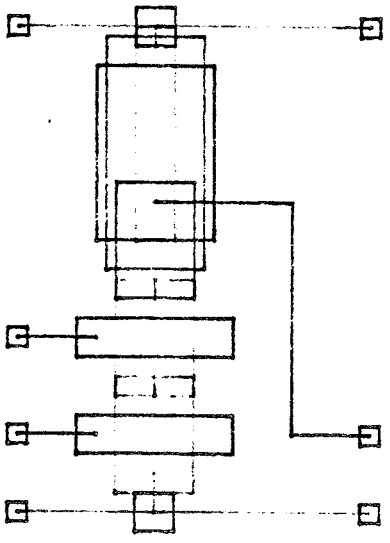


Figure 6.2. Simple Method Cannot Size NAND Structure Properly.

6.4. Better Gate Recognition Heuristics

The gates in figure 6.3 are not recognized by Andy's gate recognition algorithm, the first because the pulldown structure is not tree-like, the second, because the pulldown structure does not pull directly to ground, but rather to other signals. They are valid nMOS circuits that should be recognized and handled properly. Recognition of the graph structure pulldown on the right side of figure 6.3 is relatively straightforward, since the algorithm could notice that both parts of the pulldown include the same transistor. However, the exclusive-NOR on the left is more difficult, primarily because part of its pulldown structure is in the gates that drive its inputs. This circuit is similar to the select logic transistors in memories, which act at some times like pass transistors and at other times like pulldown transistors.

6.5. Should the Gates Be Described in the Data Format?

Why is it necessary to recognize gates at all? The Sticks Standard is used as a symbolic interchange form for all Sticks processing programs. Those program do not have to recognize Sticks components every time a file is read. Similarly, Andy could read and write an *electrical symbolic* form, a form that expresses complete electrical entities. Currently, Andy goes halfway to an electrical symbolic form, since it outputs a text representation of the node and gate data structure. A simple modification to read such a form would seem a reasonable alternative to deriving all that information every time the cell is read.

The major problem with stating gates in a data format is that the

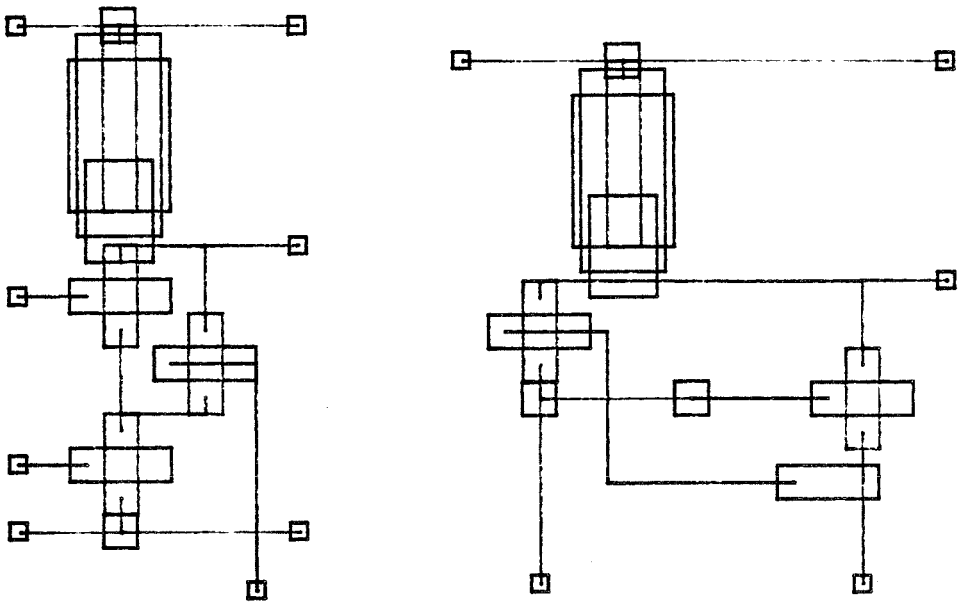


Figure 6.3. Two Common Ill-Formed Gates

decomposition of a chip into gates is rarely the same as the decomposition into cells. Cells are commonly parts of gates, and gates often span large distances on the chip. An example of the independence of the design hierarchy from the gate decomposition is the PLA. The PLA is typically composed of cells that optionally include one transistor in a large fan-in NOR gate. The question is where to put the gate, since none of the cells contain the whole thing.

If each cell contains its own part of the gate, then the gate recognition algorithm must still be run to resolve the problems at boundaries of cells. Actually, it is impossible to specify the gates in a cell, as shown in figure 6.4. The cell *tr* is used in two radically different ways in the cell, once as a

pulldown transistor and once as a pass transistor. It is impossible to characterize the transistor as anything other than a simple transistor, which is what is done in the Sticks form.

The opposing argument states that it should be illegal to specify incomplete gates, as shown in figure 6.4, just as designers using Sticks cannot make individual mask changes. It also seems that all one need do to find a gate structure is smash the lower levels of the design hierarchy. Although this argument holds for many designs, there are some designs that resist gate-level categorization. Indeed, it would seem unwise to restrict designers to gates when much of the design cannot be categorized as gates.

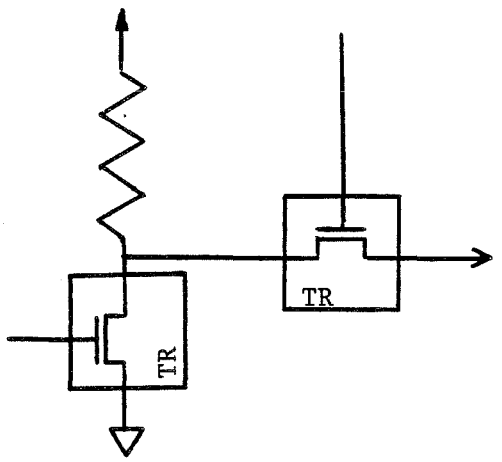


Figure 6.4. The Transistor Cell Gate Structure Cannot Be Known in Advance

6.6. Problems With Unsorted Paths

The longer delay paths are sized first so that there is no chance of accidentally lengthing a path beyond the critical path. A situation where a path might otherwise be lengthened beyond the critical path length is shown in figure 6.5. If the shorter of the non-critical paths is sized first, then re-sizing for the longer path may cause the longer path to be longer than the critical path, since gates cannot be made that generate results in negative time.

The dangers of power optimization without sorting paths includes not only making delays longer. A gate sizing algorithm that attempted to make the remaining gates of the longer non-critical path very fast might find a solution

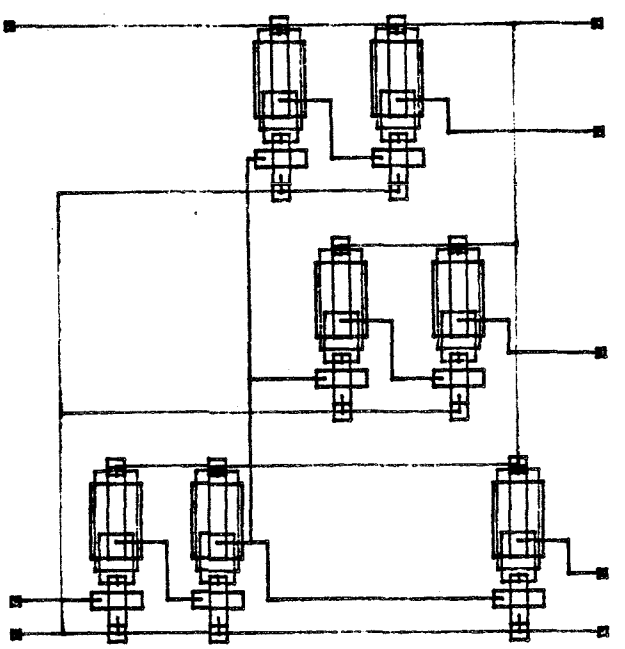


Figure 6.5. Sizing Paths Without Sorting Causes Problems.

which, although meeting delay requirements, was very wasteful of power, possibly offsetting the advantages of the rest of the power optimization.

Some of the problems with unsorted paths can be solved by simply re-sizing all gates to smaller values along the longer critical path in figure 6.5, not just the gates that had not yet been sized. In this case, the remaining gates along the shorter non-critical path would not be sized properly. They would will still be too fast, and more power could have been saved. Sorting the paths is an inexpensive and accurate solution.

6.7. Problems With the Current Path Sorting Method

The current method of sorting paths and optimizing power in order of length is not free of problems. Figure 6.6 demonstrates a case where the current algorithm can fail. First, assume that none of the paths is the critical path, and gates in both horizontal paths will be made smaller to save power. The two horizontal paths (A B C D and F G H) are longer than the short path that connects them (E H), so they will be sized before the diagonal one. This causes the gates on the two ends of the path (B and H) to be resized independently. It may therefore be necessary to choose a negative delay value for the central gate so the delay of its short path does not exceed the delay between the already-sized gates at the start and at the end of the path. The result may be an enormous gate (E) or simply an error, creating a new longest path (A B E H) that may be longer than the critical path.

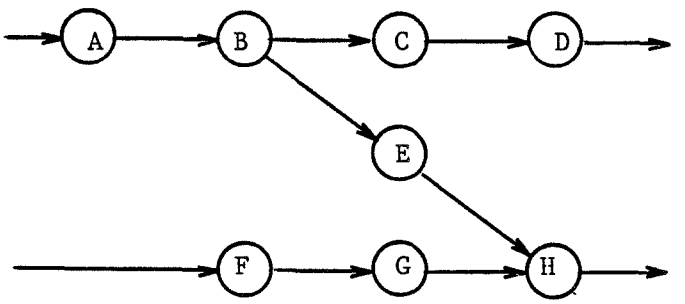


Figure 6.6. A Situation That Causes Problems With Sorted Path Optimization

6.8. Limitations of the Andy Clocking Model

The timing model used by the CLOCKing option is a rather simple one. It uses the two phase non-overlapping clocks described in [Mead 1980]. In this section, the clocking model used in Andy is compared to other sequential timing models.

A typical sequential circuit with a two phase clock is shown in figure 6.7. The circuit is composed of logic blocks that contain no state, separated by pass transistors that clock the data between the blocks. Each logic block generates results that must be ready before the clock signal following the block goes low. This falling signal indicates that the data are valid for the next logic block. In the figure, the data for logic block 2 are stable when φ_2 goes low and the results must be ready when φ_1 goes low. Therefore, the logic block must generate its results in the time between the falling edges of the clock signals. Andy uses this simple timing model for its path analysis,

since it uses pass transistors that are controlled by CLOCK signals. Andy assumes that the critical path that determines the clock frequency lies between two clocked transmission gates.

The real behavior of this situation is more complex than that which was just described. Some of the outputs from logic block 1 become valid before the falling edge of φ_2. Logic block 2 can begin calculation of the next results before the falling edge of φ_2. This more complex model requires that the optimization be carried out on paths starting at φ_1 through block 1, through the φ_2 pass gate and through block 2. The paths starting at φ_2 through block 2, φ_1 and block 1 must be optimized independently. Andy does not support this timing model.

Notice that this two-phase scheme can have precharged gates in the form shown in figure 6.8. The inverter is precharged high during φ_1 and computes its result during φ_2. The delay of the signal on the output of the gate is

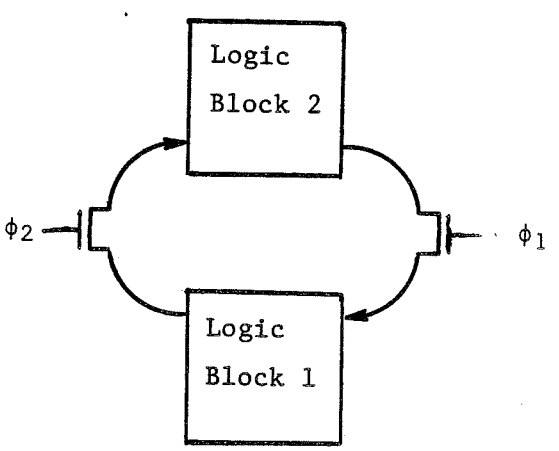


Figure 6.7. A Typical Sequential Circuit.

dependent only on the fall time of the gate. The precharging does not change the timing or the analysis. Paths of gates with precharges are legal gates in Andy, and are treated correctly, since the CLOCK nodes on transmission gates still determine the optimization paths.

The same precharge structure is used in a four-phase clocking scheme, figure 6.9. In the four-phase scheme, the precharge gates themselves provide the timing. The output of the first inverter is precharged on φ_1 and is calculated on φ_2. That signal is valid on φ_3, when both φ_1 and φ_2 are low, isolating the output node. The second inverter precharged during φ_2 when the first output result was being calculated, and the second inverter calculates its result during φ_3.

In this four-phase timing model, the gate precharge structure provides the timing information. So the precharge gates must be recognized as breaking optimization paths, like the transmission gates with CLOCK nodes in Andy.

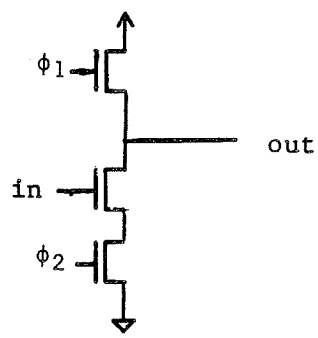


Figure 6.8. A Simple Precharge Gate

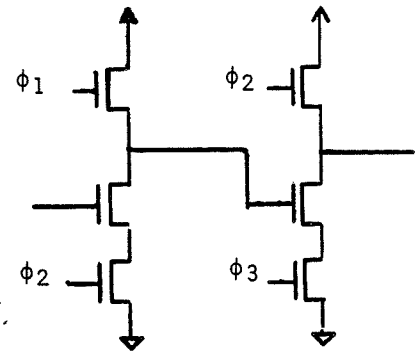


Figure 6.9. Precharge Gates in a Four-Phase Clocking Scheme.

Since these are exactly the same structures used in the two-phase precharged scheme described above, an automated process cannot tell them apart. If Andy were restricted to one of these two timing strategies, though, it could be recognized. Alternatively, the labelling of timing gates, those that break optimization paths, could be left to the user. Andy uses the former method, limiting the designs to the simple two-phase clocking model.

6.9. Non-Rectangular Transistors

Drivers of large loads frequently have non-rectangular transistors so they can form more compact structures. Some algorithm for bending or snaking very large transistors would be useful. It would seem reasonable, since choosing the shape of a transistor is motivated by area constraints, that the performance optimizer not be required to find appropriate shapes for large transistors, but the performance optimizer should be able to handle bent transistors.

6.10. Additional Constraints

Consider the case of the shared bus, where the user wants to constrain the load. Without any constraints, Andy assumes that the load is the sum of all loads on the bus. Ideally, the load should be constrained to be worst of the individual loads that the bus might drive. At the least, one would like to set the bus load to the bus parasitics plus some additional load, the expected worst-case load. The load constraint in Andy, however, only allows a single number to represent the constrained load on the bus.

Since Andy deals with delays, it would seem reasonable that the designer should constrain delays in the design. Most delay constraints are implicit in the design: it is rather obvious that all delays should be as long as the longest delay. But there are a few situations in which delay constraints are meaningful. First, one would like to set the overall delay for a circuit between clock phases. This is currently done with a command to the Andy power optimizer, rather than a constraint, but its inclusion as a constraint would not be difficult.

A second situation in which a delay constraint might be useful is when setting the delays of several signals that form a composite multiple-wire signal. An example is a sixteen-bit parallel data bus from a processor. External circuitry cannot use any bit until all are ready, so there is no need to drive some faster than others. Not only would one like all signals to be driven at the same speed, but one would like to set high and low bounds on the delays. Setting a maximum bound is not difficult, but the current algorithms would have some difficulty if the bounds were too extreme. Andy cannot make arbitrarily slow or arbitrarily fast circuits.

A third situation where a delay constraint would be useful is in setting the size of a precharge transistor. The delay to precharge the signal should not exceed the clock delay, which is the duration of the other clock phase.

6.11. More Accurate Delay Models

The delay model used in Andy is admittedly simple. More accurate models, particularly for long wires, are commonly used in path delay analysis software. The wire model used in [Putatunda 1982] provides a reasonably good estimate of wire delay. This system uses the average of the Penfield voltage bounds to get a reasonable estimate of the voltage over time. The system extracts a delay for driving the node by measuring when the voltage reaches a predefined point. The delay is divided by the effective resistance of the wire plus the driving transistor to get an effective capacitance that is used in simpler calculations, later.

Inclusion of pass transistor resistance would improve the accuracy of the delay equations considerably, and the same mechanism that included the pass transistor resistance could be used to include wire resistance also.

More complex models such as these which take into account distributed resistance and capacitance could be used in Andy, but care would have to be taken when making the changes because changes in the transistor resistance do not affect the effective resistance of the node, and hence the delay in the straightforwared way it does in the simple model now in use. Complex ad-hoc approximations like this one can have disastrous special cases. Before one is used, some work must be done to be sure that those special cases are not catastrophic.

The simple transistor model does not take into account sidewall capacitance and other more minor effects. A more exact approximation of the delay from the transistor would lead to more accurate delay calculations. Although some existing systems use the simple RC model used in Andy, others use more complex models. The table in figure 6.10 shows a few that have been in the literature recently. These equations are all empirically derived. Therefore, it is unknown if these equations would be valid in a general case.

A problem with more complex delay equations is that some equation must be used to translate backward from the capacitance on the node to a desired resistance of the pulldown transistor. It is not immediately clear with some of the more baroque transistor models how to do this. Even with the relatively simple models, some work must be done first to be sure that sizing feedback loops still terminates and that there are no additional less-optimum stable sizes for feedback loops.

Finally, at this time it is not clear how much performance optimization can be gained with more accurate models. Before putting out all the effort, one should be apprised of the gain. Andy could be used as a testbed for the investigation of the advantages of these more accurate models.

[Nham 1980]	$t = kRC_L$
[Koppel 1978]	$t = k + k_1 t_{ri} + k_2 C_L + k_3 t_{ri} C_L + k_4 (t_{ri} C_L)^2$
[Putatunda 1982]	$t = k + k_1 C_L + k_2 R_L C_L$
[Pilling 1972b]	$t = kRC_L$

Figure 6.10. Some Transistor Delay Models in the Literature

CHAPTER 7

Summary

This chapter summarizes some of the philosophy about the relationship between the work reported in this thesis and Sticks symbolic layout. It starts with a description of the similarities between the performance optimization described here and Sticks symbolic area optimization. This discussion is carried to the role of symbolic systems such as the Sticks and Andy in supplanting parameters on parametrized cells, and the role of such parameters in defining the design hierarchy. Later sections discuss the role of a performance optimization system such as Andy in a complete design system.

7.1. Similarities With Sticks

The table in figure 7.1 summarizes the comparison between Sticks area-based optimization packing and stretching with Andy performance optimization and power optimization. Many of the parameters are similar, yielding similar algorithms and similar language for describing the processes. Others are rather different, and serve to make more noticeable the fundamental differences between the two operations.

Neither operation attempts to reach an absolute optimum. The optimization operation only gives a local optimum for the parameter, and it may be that a better value can be gotten by modifying the algorithm or the topology.

Attribute	Sticks	Andy
Primary optimization parameter	Length	Delay
Secondary optimization parameter	Length (other axis)	Power
Unit of manipulation	Component	Gate
Connections between units	Twigs	Nodes (signals)
Parameter manipulated	X or Y position	Device size
Search limiter	Spatial locality	Electrical locality
Invariant	Spatial topology	Electrical topology
Algorithm limits	Geometric design rules	Fanout rule, min. device size
Constrained Value	X,Y position	Capacitance, Transistor L,W

Figure 7.1. Comparison of Concepts in Area Optimization and Performance Optimization.

Both operations optimize with respect to a set of inviolate rules. The Sticks rules are minimum geometric size and spacing rules, the Andy rules are fanout factor and minimum device size rules. More lenient rules, such as a smaller line width or better metal spacing in Sticks yield better optimizations. Similarly, a different fanout factor and lower capacitance devices and wires yield faster circuits in Andy.

Andy uses an electrical locality, where adjacency is determined by electrical connections between gates. This electrical locality does not follow the design hierarchy, which is related to the physical decomposition of the design. It is common to have several elements from all levels of the design hierarchy closely related by their electrical locality. Electrical nodes must cross cell boundaries.

7.1.1. The Unit of Manipulation

There are interesting differences between the units of manipulation of the two systems. Andy's gates and nodes are much larger and much more removed from the physical layout than the Sticks components and twigs.

The Sticks components are good atomic units of the design. They are infrequently split by cell boundaries. Indeed, the requirement that the Sticks designs include only whole components has not met significant resistance among designers.

Andy's gates and nodes, on the other hand, are frequently split by the design hierarchy. It is not possible to determine what makes up a gate or what that gate must drive until the entire hierarchy is known. Since designers prefer to think in terms of the design hierarchy, Andy-like composition systems have received very little notice compared to Sticks leaf cell design systems.

This globality of gates and nodes also shows the relative importance of the performance optimizer. Sticks systems have not been embraced by designers who believe that they can envision all the design rules and design with them better than the Sticks system. When the rules that the designer must work with are all local, and do not depend on far-away, possibly as-yet-undesigned pieces of the layout, this is true. But the performance optimization cannot be done on a small, local cell. The entire hierarchy must be examined. The amount of information is just too great for a human to handle, even for a small chip.

7.1.2. Constraints

In Sticks systems, constraints are applied to the positions of components, the primary optimization parameter. In the Andy system, the constraints are applied to loading on nodes and connectors, and to device sizes. While the device size is the attribute being modified by the algorithm, the primary optimization parameter is the delay across the circuit. Indeed, when the

critical path is analyzed in the secondary optimization pass, delays, not device sizes are inserted into the graph.

Since the purpose of the constraints is to limit the cleverness of the optimizer, constrained loads and device sizes serve well. They provide an interface with which the user is comfortable and which can be seen in the representation of the circuit. As discussed in the previous chapter, there are a few cased in performance optimization and power optimization where one would like to constrain delays.

7.1.3. Tertiary Optimization

Andy performance and power optimization and Sticks area-based optimization packing and stretching can be envisioned as automated solutions to multiple criterion optimization problems. The general algorithm attempts to optimize the primary parameter in the "packing" step, then trade off optimization of that parameter when it does not affect the critical path for the secondary parameter, in the "stretching" step.

These optimizations must be applied after the composition has been specified to achieve some kind of global optimization, although some packing could be done on a cell-by-cell basis.

In Sticks area optimization, the primary parameter is the length of the side of the cell and its optimization is the familiar Sticks compaction. In the second phase, constraints are determined so that the primary parameter will get no worse, then all elements of the design that are off the critical path of the primary parameter are made worse so that all path lengths match the critical path length. This is the determination of stretch values in Sticks

followed by stretching. Length in one dimension is traded for wiring channels, saving length in the other dimension.

In the Andy operations, the primary parameter is delay propagation across the circuit. The secondary parameter is power dissipated by the chip. Notice that with Andy, we could have reversed the order of parameters and gotten the lowest-power design first (putting some constraint on pullup transistors to keep them from getting ridiculous), then trading power for speed making, perhaps, all parts of the chip dissipate the same power per unit area, regaining speed.

It is difficult to see the parameters being traded off against one another in Sticks because both the primary and secondary parameters are length. Also, Sticks systems have only been used recently for stretching. The delay-power tradeoff is much more familiar and has been worked for many years.

There is no reason to have only two parameters. There can be a tertiary parameter, and so on. During each stretch operation, all previous parameters not on their respective critical paths are made less optimal in favor of the new parameter.

This type of optimization, adjusting one parameter at a time, is rather amenable to automation, since it requires no heuristics for exploring the design space. However, it gives a design in which the critical parameter absolutely optimized, and others are truly in a subordinate role. It is often easy to see which dimension is critical in Sticks systems, so absolutely optimizing that dimension at the cost of the other is not a great loss. When trading off other parameters, for example speed, power and area, some middle ground away from "optimally fast", "minimum size", and "lowest

power" is often desirable.

The system described in this thesis is not usable directly for such optimization, but there are solutions to the problem. First, the order of the parameters may be varied in different parts of the design. Alternatively, optimizations could be allowed to reduce a previously-optimized parameter by some amount, say five or ten percent, if the savings in the current parameter was great enough. This would ease the restrictions on the secondary parameter and could result in a better overall design. This type of optimization is seen in Sticks cells in which the minimum area usually does not have one dimension totally minimized.

7.1.4. Algorithmic Similarity

The optimization and constraint generation occurs in the net defined by the electrical adjacency, which is analogous to the solution graph used in Sticks systems [Mosteller 1981] The node and gate structure graph is shown in figure 7.2 and a sample compaction graph from Rest is shown in figure 7.3. Conceptually, the graph is solved in much the same way. However, additional

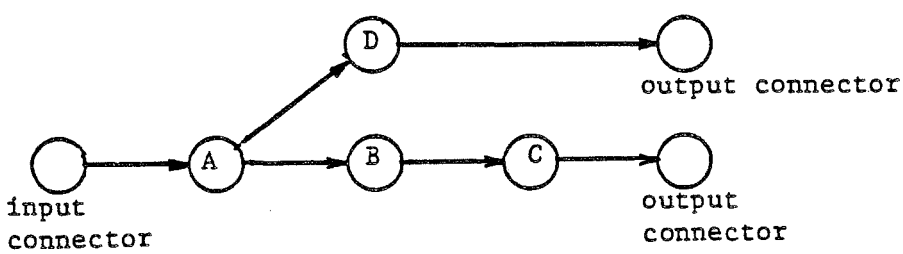


Figure 7.2. Node and Gate Data Structure.

complications in the graph solving algorithm for delay and power optimization arise because we wish to spread out the delay savings evenly among as many gates as possible in order to save as much power as possible.

7.2. Parametrized Cells and Symbolic Layout

A *parametrized cell* is a cell that is defined as an algorithm that accepts parameters. The cell can change to match its environment, reducing the number of unique cells. The parameters to cells fall into three categories:

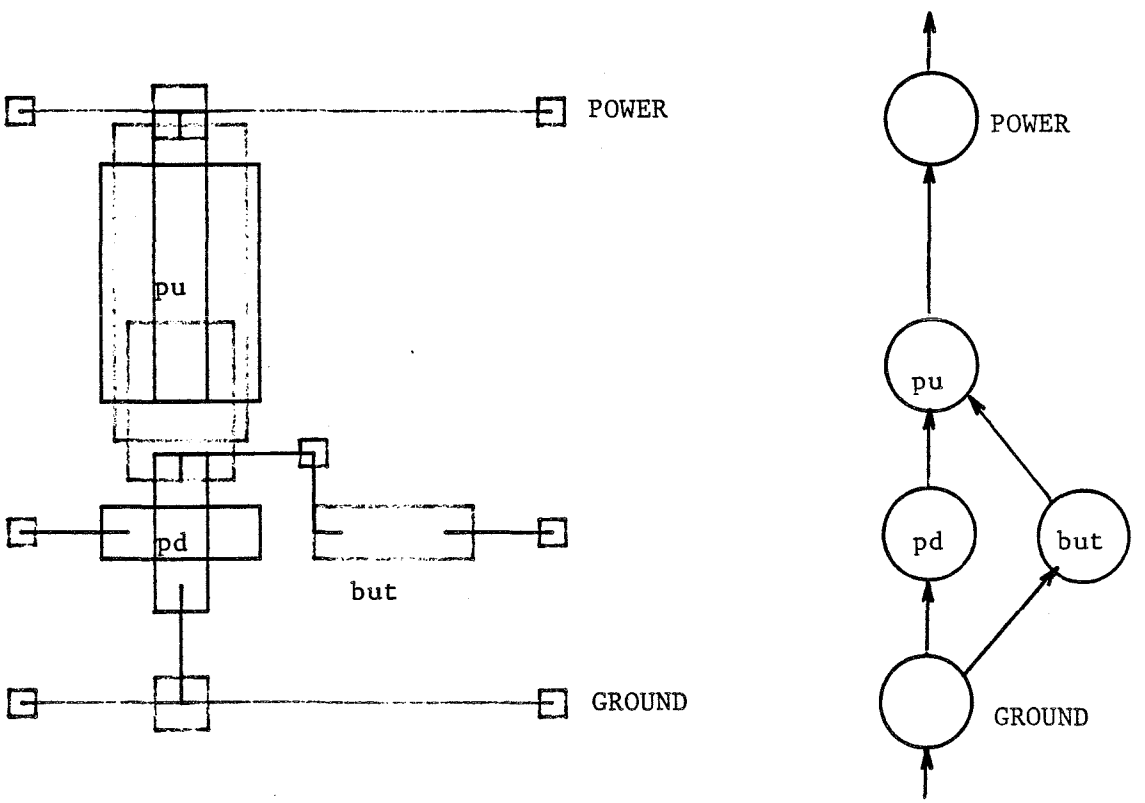


Figure 7.3. Symbolic Layout Compaction Graph

physical properties, electrical properties, and behavioral properties.

Physical properties are primarily connection points for cell stretching. The algorithm that represents the cell includes a stretching algorithm to make a geometric transformation. This algorithm allows connectors to be moved to the positions given in the parameter list while maintaining geometrical correctness.

Electrical properties include power loading and signal strengths. Electrical parameters are much less common, and are handled by the algorithm internally by sizing devices and performing a kind of stretching operation, if needed, so the cells remain geometrically correct.

The third category of parameters is behavioral properties, which includes number or bits in a data path, ROM and PLA coding, memory array sizes and conditional geometry.

Parametrized cells can accept transformations like a geometrical data form, device stretch positions, like a symbolic form, and device sizes like an electrical symbolic form. In parametrized cells, all the processing of the parameters is done by the cell. In Sticks and Andy symbolic systems, an external algorithm handles all the parameter resolution. This is reasonable for physical and electrical properties, since the way the optimization is done does not vary from cell to cell. However, behavioral parameters are very different.

Sticks eliminates the need for the user to deal physical parameters by providing a form that can be stretched and an algorithm for stretching all cells. The algorithm to modify component positions is external to the cell

and is shared among all cells. In a symbolic layout system, there is no need for processing of physical parameters, the processing is incorporated in a program that relies on the malleable positions of components.

Processing of electrical parameters is handled in a similar fashion in Andy. The electrical symbolic data form relieves the user of the electrical parameters, and there is an algorithm to perform the electrical composition and optimization that is external to all cells and applicable to all.

This leaves only behavioral parameters. By their definition, behavioral parameters are inherent to the design. Our goal is to shield the user from having to deal with too many unnecessary parameters. Since behavioral parameters cannot be eliminated, it would appear that a system that eliminated the need for the user to address physical and electrical parameters would be the best automation that could be produced. Andy is such a system.

This treatment of parameters can be seen in the Sticks Standard. Those parameters that are passed from the instance are specified in the cell by "soft" numbers. In the Sticks Standard, the only hard numbers are the device sizes. Therefore, an electrical symbolic format such as Andy's, that makes those numbers soft as well, eliminates *all* hard numbers in cells. This seems to be an indication that something is being done right.

7.3. The Relationship Between Hierarchy and the Design Data Format

Some people argue that since the electrical properties are inherently global, a system that modified them would, of necessity, destroy the hierarchy. This

statement is false on two counts. First, although electrical components may be global *spatially*, they are perfectly local *electrically*. Electrical modifications of parts of a circuit must look at other parts of the circuit, but not more than one electrical node away. We have electrical topology similar to the geometrical topology in symbolic layout systems. The electrical topology is based on electrical nodes, and modification of that topology is the creation of more nodes.

Second, destruction of the hierarchy depends on the hierarchy you view. If the design hierarchy contains elements in an electrical symbolic format, then cells can be personalized electrically without changing the hierarchy or defining new cells. If you envision your hierarchy as containing symbols with hard sizes on electrical components, like the Sticks Standard components, then electrical optimizations do indeed destroy the hierarchy, just like a Sticks system destroys a hierarchy that contains only symbols defined as absolute geometry.

The number of "unique" cells in the system depends on the definition of unique. If "unique" means "having a particular hard geometry", then symbolic stretching of a cell creates a new cell. If "unique" means having a particular set of device sizes, then electrical optimizations create new cells, but a Sticks-like positioning does not. If "unique" means having a particular set of devices, regardless of size or absolute position, then an electrical optimization does not create new cells.

Sticks systems have not be envisioned as destroying hierarchy because they have typically performed the compaction function before chip assembly. This means the designer has control of the number of different compactions

of the cell (typically one compaction to some minimum area). However, the user cannot take advantage of the stretching properties when the cell is interfaced to other cells in the system, losing the most important aspect of the Sticks form: the ability to modify the positions of components when the chip is assembled. Thus, early Sticks systems still had to route wires to match connections to the exterior of cells [Williams 1977]. This revelation also explains why assembly systems such as Riot [Trimberger 1982b] create such a large number of cell definitions -- the design tools still envision a cell as having a particular hard geometry, so differently-sized cells required different cell definitions.

An electrical symbolic system creates a large number of cells if we envision cells as primitive geometrical or topological objects, and if we insist on using the full power of the malleable-transistor form. This proliferation of cell definitions in an Andy-like system would not be apparent either if performance was optimized within the cell without considering the environment, then the cell used in designs, because the control of cell creation falls back on the designer. But in this case the designer is forced to use pre-optimized cells that may make a good implementation in the new environment. The is the same problem faced in standard cell systems that must add function outside the cell to make the logical interface.

7.4. Andy as a Piece of a Design System

Performance optimization alone cannot make a design system. A design system requires tools to generate the data Andy uses, and tools to take Andy's output and produce a chip. An example of such a system is outlined

in figure 7.4.

The optimization system takes input from any of a number of assembly tools and performs the logical composition, making physical connections to correspond to the logical connections between instances, and making the equivalent electrical composition, setting device sizes so that all signals can be driven quickly. The system optimizes area, delay and power.

Of course, all composition tasks must process the complete design hierarchy. The performance optimizer described in this thesis does so, and an area optimizer must work in conjunction with it. Another part of the system must be a program to set power wire widths to meet current density limits. Such a feature is relatively simple, given the data structure in Andy.

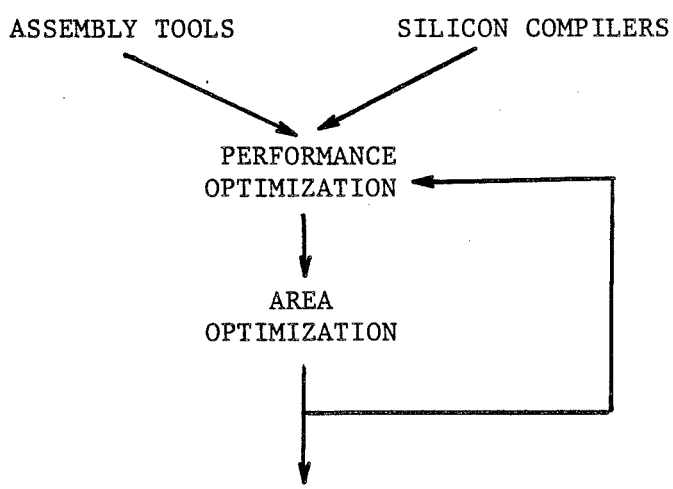


Figure 7.4. An Electrical Symbolic Design System.

7.4.1. Interaction of Optimization Tasks

The changes Andy makes in device sizes may violate geometrical design rules. Therefore, after performance optimization, the circuit must be put through a Sticks area optimizer to repair geometrical design rule violations. The changes made by the Sticks compactor will be reflected in different wire lengths, changing the parasitic capacitances and therefore the required drive of the gates. This will require another delay optimization with the new parasitic capacitances.

This iteration through performance optimization and area optimization should settle down relatively quickly, since delay and area are not tightly coupled. During delay optimization, transistor sizes increase by a factor of *1/fanout factor* in width at the most, and may actually become smaller. A segment of wire of comparable length is generated by the area optimizer with capacitance also proportional to *1/fanout factor*. This increase leads to increased device sizes of order $1/fanout\ factor^2$ to accommodate the increased capacitance of the longer wires. Since the gate sizing has a cutoff point, this iteration must coverage.

Iteration of the optimization steps may seem unsettling, but it is already done in area compaction in which X and Y dimensions are compacted independently. Since the X and Y dimensions are indeed dependent on one another, the compaction steps must be iterated. Since the size of wires and the size of transistors are related, and we wish to optimize them separately, we must iterate the optimization steps. Of course, a different order of optimization steps leads to a different final circuit.

If the area composition system has a choice of where to lengthen wires, it

would certainly be advantageous to decide which wires to lengthen by the effects of the longer wires on the delays. A reasonable choice is the minimum capacitance solution produced by the *affinity* algorithm in Rest [Mosteller 1981]. However, this is not necessarily the minimum delay solution. Rather than build a system with the delay optimization and area optimization separate, it may be advantageous to build them together.

7.5. Other Applications of Andy

The algorithm described in this thesis can be used in more than one way. It has been described as a synthesis tool for creating proper layouts. It could be used earlier in the design process with a logic diagram to determine optimum device sizes before any layout takes place. In this mode, though, it could not take into account the parasitics on the wires. Andy could also be used as an analysis tool to show where the fanout rules are not being obeyed in a circuit.

These uses of Andy could be incorporated into conventional design systems in which the circuits are described geometrically, and electrical information is extracted for analysis. It would not be reasonable to adjust device sizes in such a system because no area optimizer would be available to correct the geometrical design rule violations introduced by the electrical optimization.

However, this seems rather inefficient, because if the circuits were described in the proper form, the design system could not only check a circuit, but correct it as well.

CHAPTER 8

Conclusions

Current integrated circuit design practice does not address performance issues well. Current means of gaining better performance are expensive, generate poor optimizations or both. Hierarchical design aggravates this problem.

The integrated circuit design work in universities stresses fast turnaround and functional correctness at the expense of area and performance optimizations. The loose area design rules do not cause chips to be too much larger than those made with more precise design rules. However, many in the university community ignore performance optimization because it is difficult and because the traditional way to do performance optimization requires more design iterations. These design iterations seriously impact the delay to getting working parts.

The system described in this thesis can generate faster parts automatically. It does not require additional design iterations or costly simulation. The algorithm can be changed without modification of the underlying conceptual basis if different models or true optimum performance is desired.

The system is cheap to use in terms of elapsed time, computer time, and human operator's time. Rather than give the user statistics or telling him what to do, the system actually makes the changes that must be made to produce faster circuits.

The system has many similarities to symbolic layout. Those similarities include data structures, algorithms and general approach to optimization. Performance optimization as described in this thesis relies on symbolic layout area optimization to make design rule correct chips.

The system described in this thesis allows the designer to use more advanced assembly tools, such as stretching tools, which otherwise might generate hopelessly slow chips. It allows the composition system to do electrical composition, not merely physical composition.

Traditional tools are caught between enormous circuit complexity and the physical hierarchy that is used to address that complexity. The hierarchical design does not necessarily aid difficult composition tasks, and may make them even more difficult. Tools that address some composition tasks may fool the designer by hiding other problems. It is hoped that this work will stimulate others to investigate the new role tools must play in composition.

References

[Agule 1977] B.J. Agule, J.D. Lesser, A.E. Ruehli and P.K. Wolff, Sr., "An Experimental System for Power/Timing Optimization of LSI Chips", *Proceedings of the Fourteenth Design Automation Conference*

[Anderson 1982] J.M. Anderson, B.L. Troutman and R.A. Allen, "A CMOS LSI 16 x 16 Multiplier-Accumulator", *1982 IEEE International Solid State Circuits Conference Digest of Technical Papers*

[Bening 1982] L.C. Bening, T.A. Lane, C.R. Alexander and J.E. Smith, "Developments in Logic Network Path Analysis" *Proceedings of the Nineteenth Design Automation Conference*

[Bilardi 1981] G. Bilardi, M. Pracchi and F.P. Preparata, "A Critique and an Appraisal of VLSI Models of Computation", *CMU Conference on VLSI Systems and Computations*, H.T. Kung, B. Sproull, and G. Steele, ed.

[Chawla 1975] B.R. Chawla, H.K. Gummel and P. Kozak, "MOTIS -- An MOS Timing Simulator", *IEEE Transactions on Circuits and Systems*, December, 1975.

[Chen 1977] K.A. Chen, M. Feuer, K.H. Khokhani, N. Nan, and S. Schmidt, "The Chip Layout Problem: An Automatic Wiring Procedure", *Proceedings of the Fourteenth Design Automation Conference.*

[Cohen 1978] E. Cohen, A. Vladimirescu and D.O. Pederson, "User's Manual for Spice" University of California at Berkeley, Computer Science Department

[Daseking 1982] H.W. Daseking, R.I. Gardner and P.B. Weil, "Vista: A VLSI CAD System", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, January, 1982.

[Foderaro 1982] J.K. Foderaro, K.S. Van Dyke and D.A. Patterson, "Running RISCs", *VLSI Design*, September/October 1982.

[Harary 1972] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1972.

[Johannsen 1981] D. Johannsen, "Silicon Compilation", PhD Thesis, Computer Science Department Technical Report #4530, California Institute of Technology.

[Kingsley 1981] C. Kingsley, "Earl: An Integrated Circuit Design Language", Master's Thesis, Computer Science Department Technical Report #5021, California Institute of Technology.

[Koppel 1978] A. Koppel, S. Shah and P. Puri, "A High Performance Delay Calculation Software System for MOSFET Digital Logic Chips", *Proceedings of the Fifteenth Design Automation Conference*.

[Lang 1979] D. Lang, "LAP User's Manual", Computer Science Department Technical Report #3356, (Rev 1982 #4379), California Institute of Technology.

[Lightner 1981] M.R. Lightner and S.W. Director, "Multiple Criterion Optimization for the Design of Electronic Circuits", *IEEE Transactions on Circuits and Systems*, March 1981.

[Locanthi 1978] B. Locanthi, "LAP: A SIMULA Package for IC Layout", Computer Science Department Display File #1862, California Institute of

Technology.

[McWilliams 1978] T.M. McWilliams and L.C. Widdoes, Jr., "The SCALD Physical Design Subsystem", *Proceedings of the Fifteenth Design Automation Conference.*

[Mead 1980] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[Mosteller 1981] R. Mosteller, "A Leaf Cell Design System", Master's Thesis, Computer Science Department Technical Report #4317, California Institute of Technology.

[Mosteller 1982] R. Mosteller, "An Experimental Composition Tool", *Conference on Microelectronics, 1982*, The Institution of Engineers, Australia.

[Nham 1980] H.N. Nham and A.K. Bose, "A Multiple Delay Simulator for MOS LSI Circuits", *Proceedings of the Seventeenth Design Automation Conference.*

[Penfield 1981] P. Penfield, Jr. and J. Rubinstein, "Signal Delay in MOS Interconnections", *Proceedings of the Second Caltech Conference on VLSI.*

[Persky 1976] G. Persky, D.N. Deutsch, and D.G. Schweikert, "LTX - A System for the Directed Automatic Design, of LSI Circuits", *Proceedings of the 13th Design Automation Conference.*

[Pilling 1972a] D.J. Pilling, P.F. Ordung and D. Heald, "Time Delays in LSI Circuits", *IEEE 1972 International Symposium on Circuit Theory.*

[Pilling 1972b] D.J. Pilling and J.G. Skalnik, "A Circuit Model for Predicting

Transient Delays in LSI Logic Systems", *Sixth Asilomar Conference on Circuits and Systems.*

[Putatunda 1982] R. Putatunda, "Auto-Delay: A Program for Automatic Calculation of Delay in LSI/VLSI Chips", *Proceedings of the Nineteenth Design Automation Conference.*

[Rowson 1980] J.A. Rowson, "Understanding Hierarchical Design", PhD Thesis, Computer Science Department Technical Report #3710, California Institute of Technology.

[Ruehli 1977] A.E. Ruehli, P.K. Wolff, Sr. and G. Goertzel, "Analytical Power/Timing Optimization Technique for Digital System", *Proceedings of the Fourteenth Design Automation Conference.*

[Sproull 1980] R. Sproull and R. Lyon, "The Caltech Intermediate Form for LSI Layout Description", from [Mead 1980].

[Sutherland 1979] I.E. Sutherland, C.E. Molnar, R.F. Sproull, J.C. Mudge, "The Trimosbus", *Proceedings of the Caltech Conference on VLSI*, C.L. Seitz, ed.

[Trimberger 1980a] S. Trimberger, "The Proposed Sticks Standard", Computer Science Department Technical Report #3880, California Institute of Technology.

[Trimberger 1980b] S. Trimberger, "Paul -- Stick Diagram Maker", Computer Science Department Display File #3898, (Rev. 1982 #5009), California Institute of Technology.

[Trimberger 1982a] S. Trimberger and C. Kingsley, "Chip Assembly Tools", *Proceedings of the 1982 International Symposium on Circuits and Systems.*

[Trimberger 1982b] S. Trimberger and J. Rowson, "Riot -- A Simple Graphical Chip Assembly Tool", *Proceedings of the Nineteenth Design Automation Conference*

[Williams 1977] J.D. Williams, "Sticks -- A New Approach to LSI Design", M.S. Thesis, Massachusetts Institute of Technology.

APPENDIX A

Andy User's Manual

A.1. Introduction

Andy is a program that takes a logical composition specification for an nMOS circuit and performs the electrical composition, which involves three tasks. Most importantly, Andy improves the speed of the circuit by adjusting transistor and resistor sizes to match the capacitive loads on them. In addition, it ensures proper pullup-pulldown ratios on all gates including those that have some inputs gated by pass transistors. Andy also flags dangerous, probably illegal conditions, such as the case where signal on the gate of a pass transistor has itself been gated by a pass transistor.

The role of Andy in the design tool structure can be seen in figure A.1. Chip assembly tools are used to specify a composition for a chip. Then area and delay optimizations improve the design. The results of the optimization steps may lead the designer to improvements that require another design cycle. The design iterates through editing and optimization steps until the designer is satisfied with the result.

The current design system using Sticks is shown in figure A.2. Andy reads and writes Sticks Standard files that may be prepared by REST, Paul, Riot, Rcomp, PLA, or other leaf cell and composition tools.

The area optimization makes the logical connection, whether by routing or

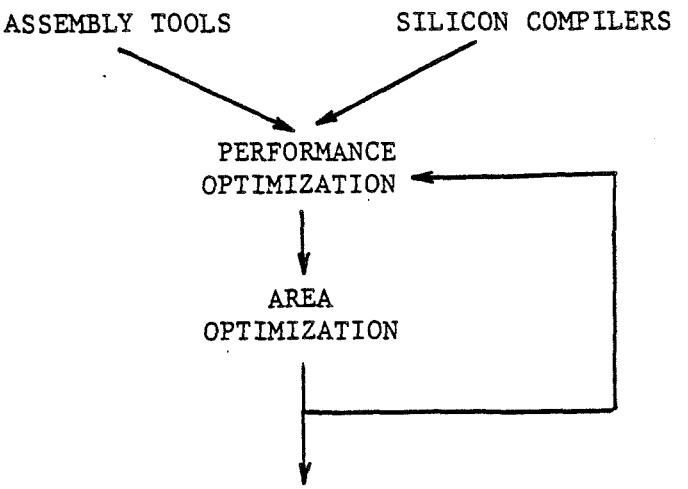


Figure A.1. Design Process Flow

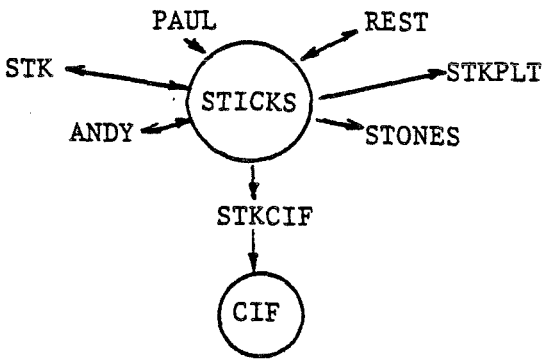


Figure A.2. Andy in the Caltech Design World

stretching, and guarantees some local optimum for the resulting size of the cell. Likewise, performance optimization makes the electrical connections, ensuring that these connections are correct in an electrical sense -- nMOS ratios are correct, and all gates achieve some local optimum for delay and

power.

Figure A.3 shows the optimization algorithm block diagram. First, Andy reads a Sticks file and extracts the node and gate data structure. Then performance optimization is done followed by the power optimization step. In the end, Andy writes a Sticks Standard file.

The delay and power optimization in Andy is a purely electronic method, dealing only with the electrical capacitive attributes of the circuit. Andy optimizes performance of an integrated system by altering device sizes to match the loads on them. Andy also makes proper pullup/pulldown ratios and fixes gate ratios for gates whose inputs went under pass transistors. Proper ratios are a side-effect of the gate sizing algorithm.

There are many other methods of performance enhancement that could be used: wires could be shortened, logic stages could be inserted or deleted to make the fanout factor as close to optimum as possible, duplicate logic could be introduced to avoid fanout. These changes are considered design issues to be handled by the designer, as opposed to layout issues that are handled by the design system. The output from Andy will direct the designer to make

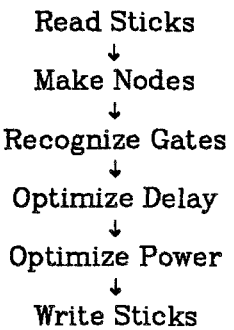


Figure A.3. Performance Optimization Flowchart

these kinds modifications of the logic to further improve the performance.

This document is the User's Manual for Andy. It includes information on the kind of input Andy expects and the kinds of operations that can be performed on that input. The document is divided into three parts: a description of the input required by Andy, a description of the commands, and a description of the algorithms.

A.2. Overview of Andy

Andy is a program that optimizes delays in circuits that are defined in a symbolic notation. The interface to the optimizations is the major facility in Andy. The Andy is a command-oriented design aid. The Andy program allows the user to read Sticks files alter then and run the performance optimization on them. The optimizations can be run independently or as a group and the user may view the result or get statistics on the resulting circuit. When the user is content with the design, he may write it back in Sticks form.

Besides an interface to the performance and power optimization algorithms, Andy has several utility functions for altering Sticks cells, to prepare the design for the optimization, and to direct the optimizations. These utilities add parameters to connectors and constraints on components and twigs in Sticks Standard cells. Andy has no Sticks editing facilities. Changes in the circuit must be done with some other tool.

A.3. The Andy Node and Gate Model

The circuit is made up of gates that drive capacitive loads on electrical nodes. A node is a collection of all the Sticks twigs and component references that are always at the same electrical potential (after everything settles down). Nodes may cross the boundaries of the physical hierarchy.

Gates are recognized on the entire cell submitted for optimization. The algorithm follows nodes across cell boundaries if necessary and moves up and down the design hierarchy to extract the gate information.

In nMOS circuits, there are basically two kinds of gates: *restoring logic gates*, with a pullup device and a pulldown structure, and *transmission gates* which are pass transistors (figure A.4). The former are unidirectional and are the form most often envisioned as gates in circuits. These unidirectional gates are made up of a single pullup device connected to the POWER node on one side and the output node on the other, and a tree-like pulldown structure connected between the output node and GROUND. A transmission gate is formed by a transistor that is not along a path from POWER to GROUND. This is the same distinction used in the gate extraction algorithm for the MOTIS simulator [Chawla 1975].

The gate recognition algorithm distinguishes between restoring logic gates and transmission gates. However, there are some MOS structures that are not allowed, and some that will not result in a gate derivation that the designer wished. Gates may have only one pullup and one output. The pulldown structure must be a true tree structure with no internal connections. Examples of well formed gates are given in figure A.4, and ill-formed gates in figure A.5. The gate on the left side of figure A.5 has a

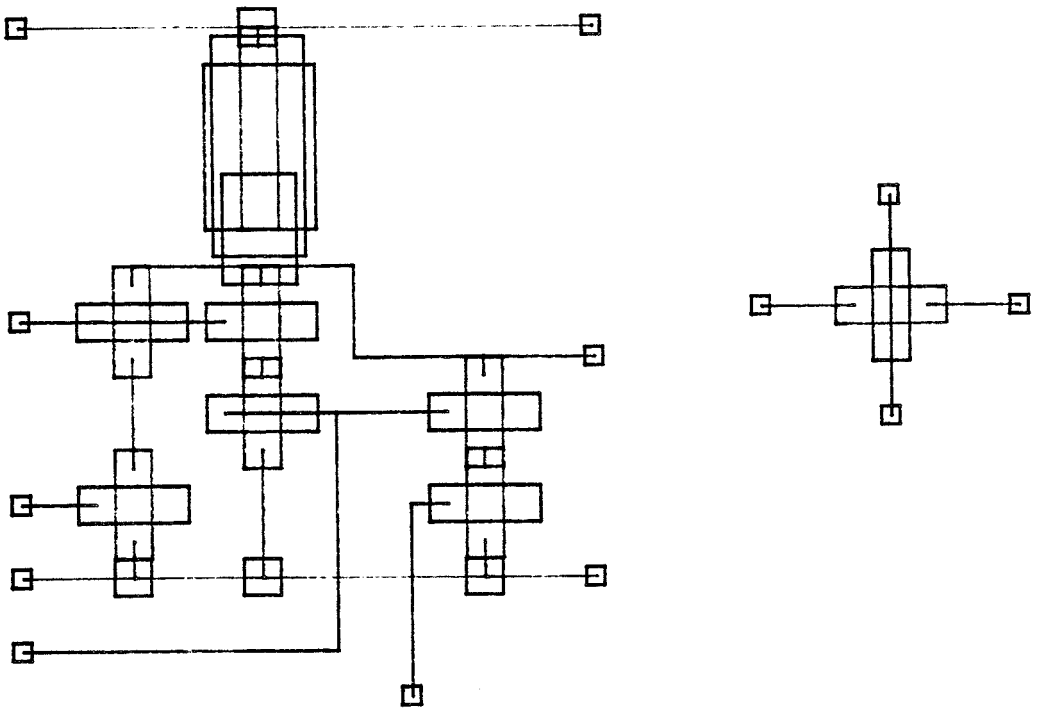


Figure A.4. Types of Gates. a) Restoring Logic Gate.
b) Transmission Gate.

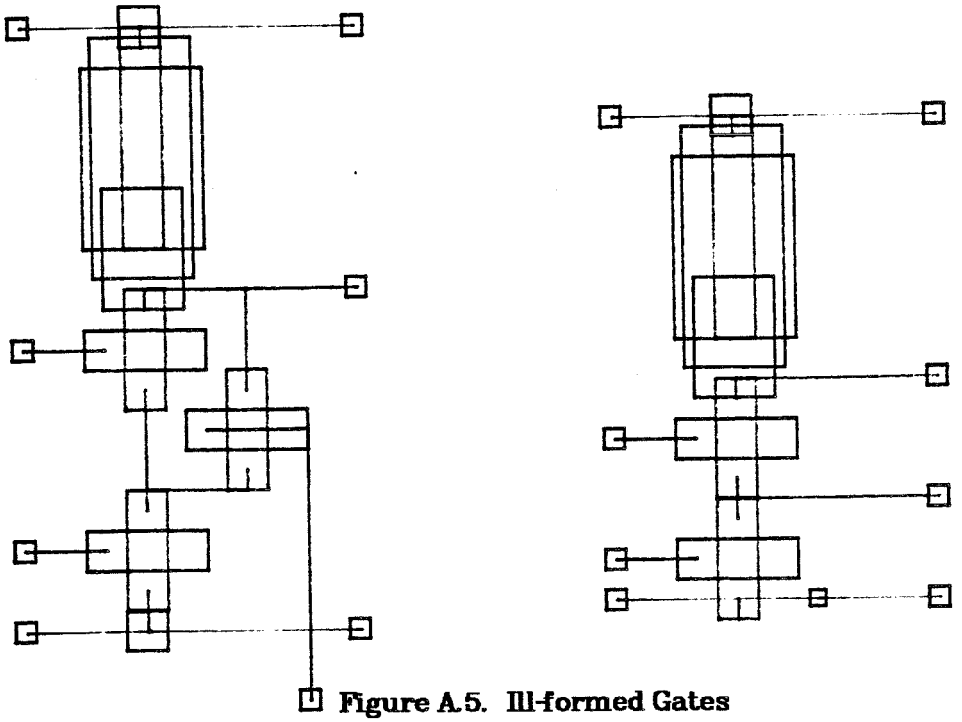


Figure A.5. Ill-formed Gates

graph-like pulldown structure. The gate on the right side has two outputs.

A.3.1. Delay Models

The delay of a restoring logic gate is proportional to the resistance (R) of the pullup times the capacitance (C) on the output node. The capacitance may include the parasitic capacitance on the wires. The delay through a chain of gates is the sum of the RC delays. This RC delay is the measure used in estimating delays in the optimization algorithms. The amount of power dissipated by these gates is inversely proportional to the resistance of the pullup.

Transmission gates are potentially bidirectional, and current supplied elsewhere will pass through a pass transistor. The optimizer attempts to keep pass transistors from being serious detriments to the performance of the circuit. It is also unreasonable to make pass transistors have a negligible effect of performance at a large cost in area. Therefore, the pass transistor resistance is set to be the same as the resistance of a pullup that would have to drive the larger of the capacitances on each side of the gate. Pass transistors are not considered in the determination of the delays in a circuit except as an additional capacitance on the node, and since they have no connections to POWER and GROUND, they do not contribute to power consumption.

There are places of special concern with bus-like structures in which the signal goes through a pass transistor. Logic on the other side of the pass transistor may at some times require that the node drive logic, and at other times the logic may drive the node. The algorithm assumes worst case in all

pass transistor situations: it assumes that it may have to drive all logic past a pass transistor at once. Therefore, the capacitance on a node that runs to a pass transistor includes the capacitance of the transistor and the capacitance on the node on the other side of a pass transistor as well. The capacitance calculation goes through all pass transistors. To limit this, the user may constrain a capacitance on a node, such as the bus node.

A.4. Input

Andy's input format is Sticks Standard with some extensions for dealing with electrical properties of the design, and some restrictions on the kinds of constructs that are acceptable so that no ill-formed gates exist. This section introduces the Sticks Standard and lists the extensions to the Sticks Standard for Andy.

Andy deals with symbolic layout defined in the Sticks Standard [Trimberger 1980a] and uses the Sticks nMOS components [Kahle 1981]. Andy accepts the full Sticks Standard including the design hierarchy. Therefore, the Riot output, when converted to Sticks form is an acceptable form for Andy. It should be noted, however, that because Andy makes changes to device sizes, there must be a compaction (area optimization) step after the performance optimization. Due to limitations in current tools, this is not possible with all circuits at this time. It seems possible, though, that RCOMP could be made to do this.

There are also several added parameters on components and constraints that are used to control Andy's cleverness that are not part of the usual nMOS Sticks. Some of these are necessary for Andy, some are desirable.

These extensions must be added before the optimization algorithms are run, and there are commands in Andy to do this. Problems occur with some software that does not accept the extensions that Andy requires, so with the current setup, it may be necessary to add these extensions every time through the design loop.

Figure A.6 shows a Sticks Standard representation of a cell with several of the additions required for Andy. These additions are discussed in the following sections.

Andy reads Sticks Standard format [Trimberger 1980a]. A sample Sticks Standard cell is shown in figure A.6 and a drawing of the cell in figure A.7. The Sticks form describes components, such as transistors, resistors, contacts and connectors; twigs, which are interconnection; and constraints, limits on the cleverness of the optimizing program that will optimize the data.

A.4.1. Parameters

Unaugmented Sticks Standard does not include enough information for performance optimization. Therefore, several parameters on components and constraints were added to facilitate the performance optimization. New parameters on components are shown in figure A.8. These parameters can be added to Sticks cells in Andy.

The gate finding algorithm can find pass transistors most of the time. However, some circuits, such as the shared bus in figure A.9, confuse it because Andy sees a path from both pullups through the Bus node, the pass transistors and the pulldown on the other bus driver to GROUND. By explicitly

```
CELL srcell 250 4
  COMPONENTS
    CONNECTOR T GROUND: gndl -48 -45  gndr 48 -45 ;
    CONNECTOR T INPUT: in -48 -29 ;
    CONNECTOR T POWER: vddl -48 45  vddr 48 45 ;
    CONNECTOR T OUTPUT: out 48 -29 ;
    CONNECTOR T CLOCK: clktop 8 59  clkbot 8 -59 ;
    NENH W 16 L 8: pd -20 -29 ;
    NENH W 8 L 8: ps N 0 -1 8 -7 ;
    NRES W 8 L 32: pu -20 1 ;
    NBUT: but N -1 0 28 -15 ;
    NDM: N1 -20 -45 ;
    NDM: N3 -20 45 ;
  TWIGS
    POLY(8):=   clkbot   8,-43   ps.G1   clktop;
    METAL(12):=   gndl   N1   gndr;
    DIFFUSION(8):=   N1   pd.SOURCE;
    POLY(8):=   in   pd.G1;
    DIFFUSION(8):=   pd.DRAIN   pu.DSOURCE   ps.SOURCE;
    POLY(8):=   28,-29  (   out)   but.P;
    DIFFUSION(8):=   pu.DRAIN   N3;
    DIFFUSION(8):=   ps.DRAIN   but.D;
    METAL(12):=   vddl   N3   vddr;
  CONSTRAINTS
    in.Y=out.Y;
END

CELL sr 250 4
  COMPONENTS
    srcell :   sr1 48 0;
    srcell :   sr2 144 0;
    CONNECTOR T GROUND: gndin 0 -45  gndout 192 -45 ;
    CONNECTOR T POWER: pwrin 0 45  pwrout 192 45 ;
    CONNECTOR T INPUT: input 0 -29 ;
    CONNECTOR T OUTPUT C 10: output 192 -29 ;
    CONNECTOR T CLOCK: clktop1 56 59  clkbot1 56 -59 ;
    CONNECTOR T CLOCK: clktop2 152 59  clkbot2 152 -59 ;
  TWIGS
    Metal :  =  sr1.gndr sr2.gndl;
    Metal :  =  sr1.vddr sr2.vddl;
    Poly :  =  sr1.out sr2.in;
    Metal :  =  pwrin sr1.vddl;
    Metal :  =  pwrout sr2.vddr;
    Metal :  =  gndin sr1.gndl;
    Metal :  =  gndout sr2.gndr;
    Poly :  =  input sr1.in;
    Poly :  =  output sr2.out;
    Poly :  =  sr1.clktop clktop1;
    Poly :  =  sr2.clktop clktop2;
    Poly :  =  sr1.clkbot clkbot1;
    Poly :  =  sr2.clkbot clkbot2;
  CONSTRAINTS
END
```

Figure A.6. The Sticks Standard Representation of a Shift Register Segment.

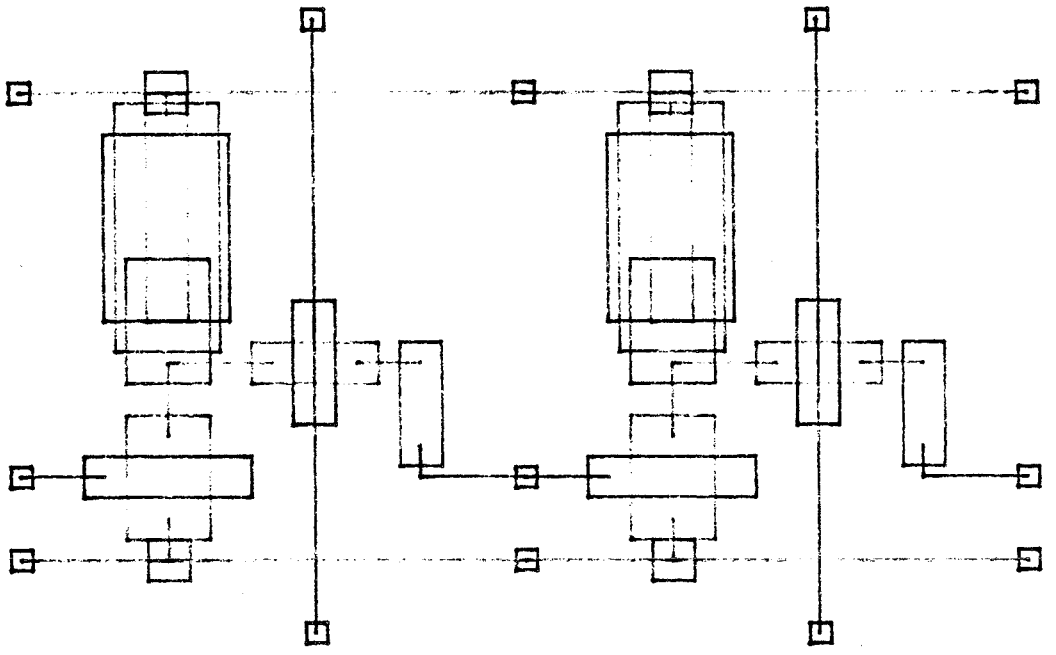


Figure A.7. The Shift Register Segment from Figure A.6.

On a Transistor	
P	The transistor is forced to be a pass transistor.
On a Connector	
T <type>	The type of a signal on the connector.
C <number>	A default capacitance on the connector.
O <number>	A default capacitance on the connector.
P	The signal on the connector came under a pass transistor.

Figure A.8. Table of Additional Parameters on Sticks Components

declaring the pass transistors in these cases or by constraining the bus node (see below), the gate finding algorithm will succeed and performance optimization will produce better results.

The type of a connector is vital to the device recognition and performance optimization algorithms. The types understood by Andy are shown in the table in figure A.10. Note that the types are all capitals.

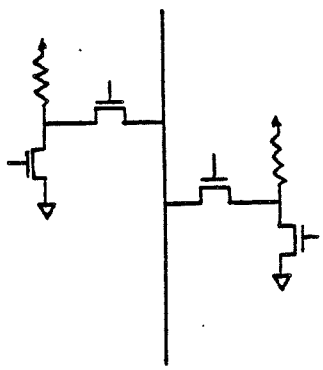


Figure A.9. Shared Bus Structure.

POWER	Power connection from the power supply.
GROUND	Ground connection from power supply.
INPUT	Signal generated outside the cell driving logic inside the cell.
OUTPUT	Signal generated inside this cell driving logic outside the cell.
IO	Signal that acts as both INPUT and OUTPUT.
BUS	Functionally equivalent to IO.
CLOCK	Signal that delimits ends of time phases.

Figure A.10. Table of Connector Types Used in the Sticks Standard

The required connector types are POWER and GROUND. If POWER and GROUND are not specified, the gate recognition will not be able to find gates in the circuit. INPUT and OUTPUT connectors may be labelled to direct the algorithm's attention. Unlabelled connectors are assumed to be IO. OUTPUT and IO connectors may have an additional parameter to simulate a load of a given number of minimum-sized transistors on the output. This simulated load is used when the cell is not used as an instance in a larger circuit, so there is no real load on the connector.

For delay calculation, every INPUT is assumed to be driven by a gate that is smaller than its load by the fanout ratio, or by a minimum size transistor, whichever is larger. Also, an INPUT connector is assumed to represent a restored logic signal unless it is marked that it came under pass transistor.

Connector types, capacitances and unrestored signal markings are only used on connectors on the cell on which the performance optimization is being done. Connectors on instances in the hierarchy are absorbed, and their attributes are extracted from the circuit.

A.4.2. Constraints

Andy uses some additional constraints beyond the simple geometrical constraints described in the Sticks Standard document. These constraints limit the performance optimizer, and are summarized in the table in figure A.11.

Andy modifies transistor lengths and widths, and the user has the ability to restrict that resizing on specific transistors. A pre-defined capacitance that is applied to a twig is transferred to the node that includes the twig when the node creation is done. This constrained capacitance then takes precedence over the capacitance that is calculated for the node. This capacitance constraint is useful in shared bus situations where the designer knows that each driver need not drive all loads off the bus at once. The performance optimizer will otherwise assume the worst, looking through pass transistors pessimistically, unless the node capacitance is constrained.

The gate finding algorithm terminates at a BUS node. Andy's gate recognition

trans	.L = <number>	The length of a transistor.
trans	.W = <number>	The width of a transistor.
twig	.C = <number>	A pre-defined load capacitance on a twig.
twig	.B	The twig referenced is on a BUS-type node.

Figure A.11. Table of Additional Sticks Standard Constraints

algorithm normally follows nodes to GROUND, which is incorrect in many cases with shared busses, such as the one in figure A.9. The BUS constraint on a twig will cause the node that contains the twig to be a BUS node, so the pass transistors that connect modules to the bus will be recognized as pass transistors not as part of a pulldown structure that extends through the bus.

Improper use of these constraints can cause the performance optimization to give wildly inaccurate results, so they should be used sparingly.

A.5. Andy Commands

This section deals with the commands to Andy. The commands are grouped into categories, and each command is treated separately. Commands, file names, and cell names are not case-sensitive. That is, the capitalization is not important. However, component names and connector types are case sensitive, so you must type them exactly as they appear in the file. When the user must type a number, if the number is a physical size, the units are lambda, as defined by the scaling parameters on the cell definition. If the number is a capacitance, the units are minimum-sized transistor loads (.01pF). If the number is a delay, the units are in terms of the resistance (as measured by a transistor length/width ratio) times a capacitance in picofarads. These are the same units put out by Andy.

A.5.1. Input and Output

Andy reads and writes the Sticks Standard, and recognizes the extensions described above. In addition, Andy can write a dump of its internal form including the node and gate information that was derived from the Sticks.

get <filename>

This command reads a Sticks Standard file from disk into memory. The file may contain many cell definitions and may describe a hierarchy for the design. The extension .STK is default.

put [cellname] [filename]

This command writes the cell into the file in Sticks Standard. The cell and all of the defining cells for the instances in it are written into the file. The extension .STK is assumed for the file. If the file name is not specified, the output is put on the terminal. The cell name is searched as defined in the Sticks Standard, first in the definition in which the user is currently working, then up the definition tree to the top-level cells. If the cell name is not specified, then the last cell that was used in any command is used.

dump [cellname] [filename]

This command writes the cell into the file in dump mode. The entire data structure including the full internal component and twig structures and all nodes and gates are written out. Only the specified cell is written, not the defining cells for the instances in it. The extension .DMP is assumed for the file. If the file name is not specified, the output is put on the terminal. If the cell name is not specified, then the last cell that was used in any command is used. If nodes and gates have not yet been recognized on the cell with the **makegates** command or some optimization command, the node and gate sections of the dump will be empty.

dumpgates [cellname] [filename]

This command writes only the gates in the cell into the file in dump mode. The extension .DMP is assumed for the file. If the file name is not specified, the output is put on the terminal. If the cell name is not specified, then the last cell that was used in any command is used. If gates have not yet been recognized on the cell with the **makegates** command or some optimization command, the gate dump will be empty.

A.5.2. Cell Management

Andy maintains a list of currently-defined cells. In an interactive system such as this one, cell management facilities are required to help the user select the cells to be optimized. Andy has facilities for listing cell names, entering a cell to view the cells defined within it, and clearing the list of cells.

list or **cells**

This command types on the display the cells currently in the cell list. If the user has **push**ed into a cell, then that cell's list is displayed. The cell names and bounding boxes are displayed on the screen.

clear

Remove all cells from the list of cells.

push <cellname>

The Sticks Standard allows cells to be defined locally to another cell. In order to view them, the user must change his cell context to that cell definition. When a cell name is specified in some other command, the search for the definition of that cell proceeds as defined in the Sticks

Standard, first in the definition in which the user is currently working, then up the definition tree to the top-level cells. Note that this does not affect the *current* cell, the cell that is the default when none is specified.

pop

This command sets the working cell to the cell that includes the current working cell's cell definition. It moves up the cell definition hierarchy.

who [cellname]

This command sets the current cell (not the same as the working cell). The current cell is the cell that is used of no cell is specified. If no cell name is given, then the cell is not changed. The name of the current cell is typed out.

A.5.3. Plotting

It is often necessary to view the data to add constraints, to understand what the optimization has done or to identify the places where the design should be modified so more optimization can take place.

Andy has a complete plotting package that includes cell selection, windowing, output device selection and scaling of the plot. There are options on plotting that enable the user to plot only the cell bounding box and connectors, and to optionally include component names on the plots. The user may plot the cell as a symbolic Stick diagram or as an abstract gate bubble diagram, showing the connections from the connectors on the cell and the connections between gates.

user <l> <b> <r> <t>

This command sets the left, right, top and bottom of the *user coordinates*, the corners of the screen in the plotting data space. The default is -1000 -1000 1000 1000.

virt <l> <b> <r> <t>

This command sets the left, right, top and bottom of the *virtual coordinates*, the corners of the area on the output device where the plot will fall, assuming the output device extends from -1 -1 to 1 1 (the square may be chopped at the top and bottom or left and right, depending on the aspect ratio of the output device. The default is the entire plotting area.

interface [cellname]

This command plots the cell bounding box and connectors. If the **names** plotting flag is on, the connector names will be plotted also.

plot [cellname]

This command clears the display then plots the cell as Sticks with the current user coordinates.

fit [cellname]

This command clears the display, sets the user coordinates to be slightly larger than the bounding box of the cell, then plots the cell as Sticks.

plotgates [cellname]

This command clears the display, sets the user coordinates to the bounding box of the cell, then plots the gates in the cell as a bubble diagram, with one bubble per gate and a line representing electrical connections between gates.

dev or device <devname>

This command sets the type of the output device. Default device is VT52. Legal devices are:

charles	Charles Terminal.
gigi	DEC GIGI Terminal.
hp	HP7221A Plotter.
7220	HP7220 Plotter.
tek	Tektronix Terminal.
tty or vt52	DEC VT52 equivalent text terminal.

names

Toggle name plotting flag. Default is OFF. When the name plotting flag is ON, all component names or gate names are plotted on plots.

half

Toggle the half-page HP plotting flag. Default is ON. When half-page plotting is ON, the HP plotters will plot on an 8½ by 11 inch page.

topqtr

Set the HP plotter to plot on top half of 8½ by 11 inch page. The device must already be set to the HP plotter.

botqtr

Set the HP plotter to plot on bottom half of 8½ by 11 inch page. The device must already be set to the HP plotter.

midqtr

Set the HP plotter to plot on middle section of 8½ by 11 inch page. The device must already be set to the HP plotter.

gatecircle [circlesize]

Set the size of the circles in the gate plot bubble diagrams. Default is 1000.

A.5.4. Stick Modification Utilities

There are two major alterations that a user must perform on the Sticks data in Andy. First, connectors must be labelled with types and given default loading. Second, constraints must be added to limit the optimization process. Constraints include loading constraints and transistor size constraints. The types and constraints are described above.

These constraints can be expressed textually, if the name of the component is known. This may not be easy if the Sticks cell was generated automatically, so Andy also provides a graphical means of identifying components. One can point to components after the cell has been plotted and set the name, connector loading, and transistor length and width. Also, constraints can be made on components. Unwanted constraints can be removed.

load <name> <type> <number>

Set the load on a specific connector. The units are minimum-sized transistor loads (.01pF). The default load for all connectors is set with the **connload** command, in the section on "Parameters to the Optimizations", below.

type <connname> <type>

Set connector type for the connector connname in this cell to the type specified in the command. Any type name is legal, but Andy only handles the ones listed above. Note that the capitalization must be the same (all caps!).

con <name> <type> <op> <number>

Make a constraint of the given type in the current cell. For example, **con foo X>2** constrains the X-value of component foo to be greater than 2. Capacitance constraints can be made also. All capacitances are in units of a minimum transistor load (.01pF).

rem <name> <type> <op>

Remove a constraint of the given type from the current cell. For example, **rem foo X>** removes the constraint given above.

set [cellname]

This command enters the **set** mode that allows the user to set parameters and make constraints graphically. Before giving the command, you must **plot** or **fit** the cell. The set command will work even if you don't, but you won't be able to see what you are doing. Because

the set command requires input from a pointing device on the output device, you must be at a Charles terminal with a mouse or at a GIGI with a BitPad.

When you are in **set** mode, you point at a component with the mouse or tablet or whatever. You will then get a prompt that gives you the following sub-commands:

name <nam>

Set the name of the component.

type <typ>

If the component is a connector, then set its type.

cap <real>

If the component is a connector, then set its default capacitance.

width <real>

If the component is a transistor, then set its width.

length <real>

If the component is a transistor, then set its length.

con <type> <op> <other>

Make a constraint. The constraint is made of type <type>, which may be **X**, for a x-dimension constraint, **Y**, for a y-dimension constraint, **C**, for a capacitance constraint, **P**, to constrain a transistor to be a pass transistor or to constrain a connector to have a signal come under a pass transistor. The <op> is the operator, which is ignored in a "P" constraint (but which must be present anyway), is one of >, <, or =, as described in the Sticks

document. <other> may be an edge constraint, **LEFT**, **RIGHT**, **TOP**, or **BOTTOM**, as described in the Sticks document, the name of another component, a number for numerical constraints, or **&**, which lets you point at another component that is to be the other part of the constraint. When the constraint command is done, the constraint is printed on the terminal.

help or **?**

This command causes a terse command summary to be printed.

refresh

Re-draw the cell on the screen.

quit

Return to Andy main command mode.

p

Proceed. This command lets you point at another component.

A.5.5. Parameters to the Optimizations

The delay and power optimizations use several global values for critical parameters. The user may set these values and thereby direct the overall operation of the optimization algorithms.

The user may turn off and on the inclusion of capacitance on wires. The wire capacitance is usually on, because it is a significant load in most circuits. The user may also control whether or not CLOCK nodes on pass transistors will break paths during delay calculation and power optimization. Turning it on allows optimization for minimum clock cycle, turning it off allows

optimization for minimum delay through a pipelined processor.

The user may adjust the most important number in the performance optimization, the fanout factor. The fanout factor is the number of minimum transistor capacitances that should be driven by a minimum transistor. The fanout factor says in some sense how concerned the user is with power versus delay. Larger fanout factor means greater delay but lower power. It may be set to any value greater than one, and is set initially to four.

The user may also change the default loading on a connector. Is is usually not reasonable that connections to the outside world have no capacitance on them. It is possible to put a specific load on a specific connector, and it is also possible to put a default load on all other connectors.

dotwigs

Toggle the twig capacitance flag. The default is ON. When the twig capacitance flag is ON, the capacitance of twigs is included in the calculation of loads on nodes.

doclocks

Toggle the clocking flag. The default is OFF. When the clocking flag is ON, pass transistors that have CLOCK nodes on their gates break paths for the power optimization. Therefore, delay and power can be optimized for either the delay through the whole cell (minimum delay for a signal to pass through the cell) or just across a clock cycle (minimum clock cycle time).

scale [sf]

This command sets the fanout factor, also known as the scale down factor. It is the number of minimum-sized transistor capacitances that can be driven by a minimum-sized transistor resistance. This number should always be greater than 1. The default is 4. If the fanout factor is not given, then the current fanout factor is typed on the screen.

connload [ld]

This command sets the default minimum load on a connector. The default is 1 minimum transistor load. If the load is not given, then the current load number is typed on the screen.

status

This command prints the value of all status variables. An example follows:

```
Current cell: SR.  MBB:  -50000,-50000  50000,50000.

Scale Down Factor = 4.00E+00.
Minimum Connector Load = 1.00E-02.
Gate Circle Size = 1000.
Won't die on error.
Trace off.
Verbose trace off.
Space tracking off.
Twig capacitance on.
Clocks off.
Name plotting off.

Plotter: VT52.  User coords:  -2000,-1000  2000,1000.
Half page HP plots
```

A.5.6. Statistics

To help the user determine the quality of a design, Andy reports statistics on the cell. The user can get the delay of the critical path, a listing of the critical path, the power consumption of the chip and the product of the delay and power. The delay and power estimates from Andy are not exact because constants are ignored, and they are based on a simple RC model of delay, but one set of statistics can be compared to another to get an idea of the relative goodness of two designs.

delay [cellname]

This command prints the maximum delay across cell and the critical path that resulted in that delay. It also includes a message if the critical path has changed since the last time it was displayed. The delay is the sum of the RC time constants for all the gates in the critical path. The resistance is unscaled as the transistor length/transistor width for the pullup of the restoring logic gate. The capacitance is the sum of all capacitances, including twig parasitic capacitance if the **dotwigs** flag is on. The capacitance calculation looks through pass transistors (transmission gates) pessimistically, assuming that all pass transistors will be open when the gate is trying to drive the node. If the **doclocks** option is turned ON, then the delay calculation also follows paths that start at pass transistors that are gated by CLOCK nodes. Those pass transistors also end delay calculation paths. The following is some sample output from the **delay** command.

```
Critical Path for cell PLA C:Y1IN G:INBUF_P1C4 G:INBUF_P2C4 G:AND_P5 G:OR_P6
 G:OUTBUF_PU1C5. Delay: 9.64E-01
Critical path changed.
```

power [cellname]

This command prints the power consumption of the cell in unscaled units of transistor width/transistor length (proportional to 1/resistance of the transistor). The power consumption for the cell is the sum of all the width to length ratios of all pullup transistors in the cell.

fm [cellname]

This command prints the delay of the critical path in the cell, the power consumption of the entire cell and the product of the two:

```
Cell PLA. Delay: 9.64E-01. Power: 2.34E+01. D*P (unscaled): 2.25E+01
```

A.5.7. Constructing the Data Structure

The data structure must be built before the optimization steps, so the optimizations build the structure, finding nodes and gates, if necessary. Andy also has commands specifically to build the data structure. These commands to separately generate the nodes and recognize the gates is included primarily as a debugging tool. The node and gate extraction algorithms are described briefly below.

makenodes [cellname]

This command causes Andy to find all the nodes in the cell. Nodes span the design hierarchy, possibly including components and twigs in instances of cells contained in this cell.

justnodes [cellname]

This command causes Andy to find all the node segments in the cell. This command differes from **makenodes** because it will not merge node segments through the design hierarchy.

makegates [cellname]

This command runs the gate recognition algorithm on the cell. If the nodes have not yet been found, **makegates** finds nodes first.

A.5.8. Delay and Power Optimization

Delay and power optimization are Andy's main tasks. They can be performed separately or sequentially with a single command. Separate commands for each step are provided more as a debugging aid than as a user feature, but there may be some situations where one or the other is not desired. The delay and power optimization algorithms are described briefly below.

clearstretch [cellname]

The performance optimization and power optimization algorithms use the same code to meet constraints. This command clears the desired delays set by the power optimization algorithm.

setstretch [cellname] [real]

This command runs the part of the power optimization algorithm that sets desired delays on gates. After this part, the cell must be run through the gate sizing algorithm to set the correct transistor sizes from the desired delays. The user may supply a number for the minimum delay for the critical path, the desired delay for the cell as a

whole. If the number is absent, 0 is assumed, and all paths in the cell are made as long as the critical path.

stretch [cellname] [real]

This command does the whole power optimization step: **clearstretch; setstretch(real); sizegates;**.

pack [cellname]

This commands does the complete performance optimization step: **clearstretch; sizegates;**.

opt [cellname] [real]

This command does the complete optimization of a cell: **clearstretch; sizegates; setstretch(real); sizegates;**. Performance optimization is done before power optimization.

A.5.9. Area Optimization

Delay and power optimization change device sizes which may result in design rule violations, mandating that area optimization be performed on the cell. Andy sends simple cells to Rest to do this optimization. Rest cannot currently handle cells with hierarchy, so some other software is needed for dealing with area optimization of composition cells. An associated program, STK, can be used to remove the hierarchy so Rest can optimize area. Other commands in STK do simple area optimizations with the hierarchy. The reader is referred to the STK documentation for more information.

packx [cellname]

This command invokes REST to perform area optimization in the x-dimension.

packy [cellname]

This command invokes REST to perform area optimization in the y-dimension.

A.5.10. Debugging Aids

There are a few commands of little or no interest to users which generate trace information during the data structure construction and during the optimization stages. There is also a command in Andy to enter the SIMULA debugger for further examination of the internal structure of the program.

debug

Enter the SIMULA debugger.

trace

Toggle the trace flag. Default is OFF. When the trace flag is ON, pages of output are generated so you can follow the program's execution. A working knowledge of the code is necessary to decipher the output, though.

vtrace

Toggle the verbose trace flag. Default is OFF. When the verbose trace flag is ON, reams of output are generated so you can follow the program's execution.

space

Toggle the free pages flag. Default is OFF. When the free pages flag is ON, messages are generated during execution which tell the user the amount of free memory space. If the program runs slowly, it is often due to large memory use. This can show which parts of the program are eating large amounts of memory.

dieonerror

Toggle the flag to enter debugger when a design error is found. Default is OFF. On the test circuits, a spurious design error indicates a bug in the program. The dieonerror flag causes Andy to enter the debugger after printing the error message when a design error is found. If the flag is OFF, the message will be printed and execution will continue.

status

This command prints the value of all status variables. An example is shown above in the section titled "Parameters to the Optimizations"

A.5.11. Miscellaneous Commands

These commands do not fall into any of the categories above.

@Andy [file]

When Andy is run, if a file is given, an initial **get** is done on that file. If the file name is **$**, then Andy starts by taking commands from the file "ANDY.INI". (Note that *commands* are taken from ANDY.INI, it is *not* read as a Sticks file.)

? or help

Type a summary of the commands.

quit

Terminate Andy execution. If you continue the program from the monitor, you get right back into the Andy command loop with everything exactly as it was.

invade

Enter "space invaders" mode for a short recreation. This mode only works properly on a VT52.

A.6. Design Rules

Performance optimization can be expressed in a somewhat formal manner by defining "design rules" which the algorithm enforces and attempts to meet as closely as possible. These rules are presented as a means of explanation of the function of Andy, not as a description of the algorithm.

(1) The minimum transistor width is 2 lambda. Minimum transistor length is 2 lambda.

This rule sets the minimum gate dimensions, which determine the cutoff for making transistors smaller. These dimensions also determine when the algorithm optimizes devices by changing width rather than changing length of transistors.

(2) A pulldown structure in a gate must have at most one square transistor resistance for each *<fanout>* minimum transistor sizes of gate capacitance that are driven by the gate.

(3) A pullup resistor must have at most one quarter square depletion transistor resistance for each *<fanout>* minimum transistor sizes of gate capacitance that are driven by the pullup.

These rules comprise the gate fanout rule. Meeting these rules is the main task of the performance optimizer. No gate may drive more fanout than the fanout variable allows. Optimal delay occurs when this number is e, but it is usually between four and eight. In the Andy system, the default value is four, but it may be changed by the user. The fanout number must always be greater than one.

(4) A pullup device that is not a depletion-mode transistor with the gate tied to the source indicates that the gate driving current is four times that of a normal gate.

A transistor-like pullup must be either a precharge device or a super-buffer device. Either way, the pulldown becomes the limiting resistance in the gate. Therefore, the gate can drive four times as much load in the same amount of time as a normal gate.

(5) A pass transistor must have at most one quarter square gate resistance for each *<fanout>* minimum transistor sizes of gate capacitance that are driven through the pass transistor.

This is the pass transistor sizing rule. It makes pass transistors the

same resistance as a pullup resistor. This heuristic is included so neither the pass transistor nor the pullup resistor is the dominant resistance on the signal.

(6) Transistor gate resistances and capacitances and interconnect capacitances are assumed to be:

Transistor Capacitance	4.0×10^{-4} pf/μm^2.
Diffusion Capacitance	1.0×10^{-4} pf/μm^2.
Polysilicon Capacitance	0.4×10^{-4} pf/μm^2.
Metal Capacitance	0.3×10^{-4} pf/μm^2.
Transistor Resistance	1.0×10^{4} Ω/■.
Wire Resistance	0.0 Ω/■.

The resistances and capacitances of the elements of the design are used by the performance optimization. These numbers are taken from [Mead 1980]. The precise values of these numbers are not important, but their ratios are important, particularly the relative sizes of the capacitances for transistors and interconnect.

(7) The resistance of a transistor which has had the signal on its gate go under a pass transistor should be considered double.

This rule compensates for the lower gate voltage on the transistors driven by signals that have gone under pass transistors. The gates will be made wider.

(8) The maximum length of a pulldown is 2 lambda.

This rule places an upper limit on the resistance of the pulldown and therefore an upper limit on its delay. This keeps the power optimization from going overboard when saving power on paths that are very far off

the critical path.

These rules define an optimum delay that is not a true global optimum. The result will be a local optimum, subject to the constraints supplied by the system, the accuracy of the design rules and the model of integrated circuit performance. This is in the same sense that symbolic layout compaction achieves a local optimum, subject the the constraints of design rules and algorithmic limitations.

A.7. Description of the Operations

This section contains a brief description of each of the algorithms in Andy. It is included as an aid in understanding of Andy's capabilities.

A.7.1. Node Determination

The node determination for a cell is done in three parts. First, all the node segments in the cell are found. These node segments consist of a Sticks twig, all the components connector references on the twig, and recursively includes other twigs and component connector references on electrically equivalent connectors on the components. Node determination passes through contacts and electrically common connection locations on transistors and connectors.

```
PROCEDURE find_nodes;
FOR all twigs DO IF twig NOT already in a node then newnode.addtwig(twig);

PROCEDURE node.addtwig(twig);
IF twig NOT already in a NODE then BEGIN
   add twig to this node
   FOR each component reference in the twig DO BEGIN
      FOR all twigs DO IF the twig has a reference to
         an electrically equivalent connector on the same
         component THEN addtwig;
      END;
   END;
END;
```

Node segment determination is done for all cells that have instances in the cell in which we are doing the node determination. These node segments are collected in the cell and merged into complete electrical nodes. The merge algorithm crawls up and down the design hierarchy coalescing node segments across cell boundaries.

A.7.2. Gate Finding

As shown in the pseudo-code below, the gate finding algorithm finds gates by following the POWER node to a transistor source or drain. Since one side of the transistor is connected to POWER, it must be a pullup for a restoring logic gate, so a new gate is created with the transistor as its pullup. Although, in the usual case, the transistor is a depletion mode device used as a load resistor, other forms for super-buffer gates and precharged gates are legal as well.

```
PROCEDURE find_gates;
FOR all POWER nodes DO BEGIN
  FOR all transistors on the node DO BEGIN
    make a new gate.
      the pullup is the transistor.
      the output node is the node opposite the POWER
    FOR all paths of transistor source and drain from the output node DO
      IF the path leads to GROUND
        THEN make them pulldowns of the gate
        ELSE make them transmission gates
  END;
END;
```

The node on the other side of the transistor is the node that the gate is driving, which must be the output node of the gate. The gate finding algorithm follows that node to find the pulldown transistor structure. When a connection to the source or drain of a transistor is found, there are two possible situations: the other side of the transistor may or may not connect to GROUND. If the other side of the transistor does not connect to GROUND, the transistor is remembered and the node on the other side of the transistor is scanned recursively, building a tree-like structure pointing to the transistors. The recursion stops when the GROUND node is found or if there are no source or drain connections on the node.

If the node is the GROUND node, then all the transistors on the path from the gate's output to GROUND must form a NAND network, serial connection to GROUND in the gate. Parallel connections to GROUND make NOR-type connections. If there is no GROUND connection, the transistors along the path must be pass transistors, and a new transmission gate is made for each pass transistor.

The gate search process can also be stopped by parameters on transistors or constraints on nodes. If the transistor has been constrained to be a pass transistor, the recursion stops, the gate determination ends, and the

transistor is made into a transmission gate. If a node is found of type BUS, then the gate finding algorithm is similarly terminated. These constraints help remove confusion in some MOS structures that do not fall into the category of well-formed gates described above, but which occur frequently in designs. These structures include shared bus structures and some more exotic transmission gate logic.

A.7.3. Performance Optimization

The performance optimization algorithm works as follows:

```
PROCEDURE optimize_performance;
WHILE some gates are yet to be sized DO BEGIN
   FOR all gates DO  IF gate.known_load THEN move_into_ready_list
   IF no gates in ready list THEN move any gate into ready list
   FOR all gates in ready list DO gate.setsize
END
```

The transistor sizing algorithm maintains two lists of gates: gates that have not yet been sized and are ready to be sized, and gates that have not yet been sized but are not ready to be sized. A gate is ready to be sized when all the loads on its output node are known. Known loads are twig capacitance, output connectors, and transistor gate connections on transistors that have already been sized.

The gates in the former list are processed, setting the sizes of the transistors that make them up, depending on the load on the output node. Transistor sizes are set to *MAX(minsize, output node capacitance/fanout factor)*. When a gate is sized, it is removed from the list:

```
PROCEDURE gate.set_size;
BEGIN
  basicresistance := MAX(min_trans_size,
    const*output_capacitance/fanout factor);
  pullup.setresistance(basicresistance*longest_NAND_length*pullup_ratio);
  FOR all pulldowns DO BEGIN
    pulldown.setresistance(basicresistance);
    pulldown.driver_node.driver_gate.sized := FALSE;
  END;
  sized := TRUE;
END
```

When a transistor in a gate is sized, the gate that drives the node that drives the gate of the transistor is moved into the list of unsized gates, since its load has changed.

As transistor sizes are set, more nodes have known loads. The gates that drive these nodes nodes can then be sized and so forth. The algorithm proceeds backward from the circuit outputs through the circuit until all gates have been sized.

In a circuit with a feedback path, the loads on some gates are dependent on the size of their own transistors. These gates cannot be sized because none of the the loads on the output nodes is defined. Andy detects and breaks the loop by simply picking one gate arbitrarily and sizing it. The transistors in the sized gate are now known loads, so the gate before the chosen gate can be sized, and so on. Eventually, the optimization makes its way around the loop to re-size the first gate. This re-sizing terminates when a transistor changes size by less than five percent. A transistor that does not change much does not move the driver of its gate node into the list of unsized gates.

A.7.4. Power Optimization

The power optimization algorithm can be expressed in general as follows:

```
PROCEDURE optimize_power;
BEGIN
  find_paths;
  sort paths into decreasing order;
  FOR all paths DO BEGIN
    find first gate that has not been optimized yet;
    current_delay := delay at end of the path -
      delay at first unoptimized gate;
    desired_delay := constrained delay at end of the path -
      constrained delay at first unsized gate;
    expand_ratio := desired_delay/current_delay;
    FOR all gates between first unsized gate and end of path DO BEGIN
      gate.constrained_delay := gate.current_delay * expand_ratio;
      gate.sized := TRUE;
    END;
  END;
  FOR all gates DO set delay to constrained delay;
END
```

Power optimization is done by sorting all the paths of gates in the cell into decreasing length. A path is a chain of gates that starts at the input connectors or at a pass transistor that is gated by a CLOCK nodes (if the **doclocks** mode is turned on) and ends at the output connectors, at the input to a gate or at a pass transistor that is gated by a CLOCK nodes (if the **doclocks** mode is turned on).

Each path is treated independently in the power optimization. All gates along the beginning of the path that have already been sized with the performance optimizer are chopped off. The delay of the remaining gates is compared to the difference in delay from the beginning of the path (either the input connectors or the last gate that was chopped off) to the end of the path (the output connector or the gate at which the path stopped). All gates in the chain are made slower by the ratio between the desired delay and the

current delay.

In the end, then, all path delays are as long as the longest delay. In accordance with the rules above, though, no gate is made so slow that a pulldown transistor width is smaller than its length. So some paths may remain faster than the critical path.

The longest delay is usually the critical path delay, but it can be set by the user, so the delay of the entire cell can be set to a desired value by the power optimizer.

A.8. Command Summary

The commands to the Andy program are listed below, each with a terse summary. Values in brackets are default values, where applicable.

get <file>	get a sticks file
put [cellname] [filename]	put the sticks cell into the file (no file=terminal)
dump [cellname] [filename]	dump the cell into the file (no file=terminal)
dumpgates [cellname] [filename]	dump the gates of the cell into the file
list or cells	type the cells defined here
clear	clear the list of cells
push <cellname>	go down the cell def hierarchy
pop	go up the cell def hierarchy
who [cellname]	[set then] print the name of the current cell
user <l> <b> <r> <t>	set the user plotting area
virt <l> <b> <r> <t>	set the virtual (device) plotting area [ALL]
interface [cellname]	plot the cell bounding box and connectors
plot [cellname]	plot the cell
fit [cellname]	set the user coordinates, then plot the cell
plotgates [cellname]	set user coords then plot gate connections
dev or device <devname>	set the plotting device to one of charles gigi hp 7220 tek tty or vt52 [VT52]
names	toggle name plotting flag [OFF]
half	toggle the half-page HP plotting flag [ON]
topqtr	set HP plotter to plot on top half of 8.5x11 page
botqtr	set HP plotter to plot on bottom half of 8.5x11 page

midqtr	set HP plotter to plot on middle section of 8.5x11 page
gatecircle [circlesize]	set the size of the circles for gate plots [1000]
load <name> <type> <number>	set a load on a connector (units are trans. loads)
type <connname> <type>	set connector type for conn in this cell
con <name> <type> <op> <number>	make a constraint of TYPE (i.e. con foo X>2) NOTE – capacitance consts are #min. trans loads
rem <name> <type> <op>	remove a constraint of TYPE (i.e. rem foo X>)
set [cellname]	set parameters of components graphically. You point at the component with the mouse of tablet or whatever. You have the following sub-commands name <nam> set the component name width <real> set the width of a transistor length <real> set the length of a transistor con <type> <op> <other> make a constraint type <typ> set the type of a connector cap <real> set the capacitance of a connector help or ? get a terse command summary refresh re-draw the cell on the screen quit return to ANDY main command mode p (proceed) point at next component
dotwigs	toggle twig capacitance flag [ON]
doclocks	toggle clocking flag [OFF]
scale [sf]	set or examine the scale down factor [4]
connload [ld]	set the min. load on a connector [1]
status	print state of all status vars
delay [cellname]	print max delay across cell
power [cellname]	print power consumption of cell (units of w/l)
fm [cellname]	print unscaled delay*power product for figure of merit
makenodes [cellname]	find nodes in cell
justnodes [cellname]	find nodes in cell but don't merge them
makegates [cellname]	find gates in cell
clearstretch [cellname]	clear desired delays in all gates for full pack
setstretch [cellname] [real]	set desired delays in all gates for stretching
stretch [cellname] [real]	setstretch(real); pack;
pack [cellname]	pack cell for speed (clearstretch; pack;)
opt [cellname] [real]	clearstretch; pack; setstretch(real); pack;
packx [cellname]	pack x-dimension using REST
packy [cellname]	pack y-dimension using REST
debug	enter the SIMULA debugger
trace	toggle trace flag [OFF]
vtrace	toggle verbose trace flag [OFF]
space	toggle FREEPAGES flag [OFF]
dieonerror	toggle flag to enter debugger on design error [OFF]
@Andy $	start with input from ANDY.INI
? or help	type this message

quit	end it all
invade	enter "space invaders" mode

A.9. References

[Chawla 1975] B.R. Chawla, H.K. Gummel and P. Kozak, "MOTIS -- An MOS Timing Simulator", *IEEE Transactions on Circuits and Systems*, December, 1975.

[Kahle 1981] C. Kahle and R. Mosteller, "NMOS Sticks Standard Components", Computer Science Department Display File #4300. California Institute of Technology.

[Mead 1980] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[Trimberger 1980a] S. Trimberger, "The Proposed Sticks Standard", Computer Science Department Technical Report #3680, California Institute of Technology.

APPENDIX B

Sticks Standard Proposal

This is version 2.0 of the Sticks Standard. This version includes clarifications to the original proposal and additions to make the Standard more useful in chip composition. This version of the Sticks Standard provides a more strictly defined syntax for parameter specification which allows new parameters to be easily added. In places where the syntax does not allow for parameter specification, an optional extension field has been added. These features allow new information to be more easily incorporated into Sticks Standard files. This version of the Sticks Standard also includes an array construct for simple rectangular arrays.

This document consists of three parts: the specification of the Sticks Standard, an example of the Standard in use, and a description of the technology-dependent parts for an nMOS process.

B.1. The Sticks Standard Overview

The Sticks Standard has a descriptive rather than a procedural semantics. That is, it describes an image rather than the means for creating the image. The Standard is intended to be a means for data interchange, not a database. Many suggestions for extensions and changes can be traced to a conceptualization of the use of the Sticks Standard as a database.

A description in the Sticks Standard carries with it a set of coordinates for

all interesting locations in the Sticks data. The locations are useful for plotting, and may be used by Sticks processing programs as an initial placement for compaction. In addition to the physical locations, the cell definition may contain a list of constraints on the final positions of the points in the design. The physical locations should be considered to be suggestive, whereas the constraints are imperative.

The definition of the syntax and semantics of the technology-dependent parts of the design are separate from this definition of the "pure" Sticks Standard, but the Sticks Standard cannot be used for a given electronics technology until those technology-dependent parts are defined. At the end of this document are the technology-dependent parts for an nMOS technology.

This description of the Sticks Standard is separated into two parts: syntax and semantics.

B.2. Syntax

The syntax description consists of a formal Backus-Naur Form (BNF) description of the syntax and a discussion of some interesting features of that syntax.

Wirth's standard notation is used [Wirth 1977]: production rules use equals to relate identifiers to expressions, vertical bar for choice and double quotes around terminal characters. Curly brackets indicate zero or more repetitions, square brackets indicate optional features and parentheses are used for grouping.

The formal syntax description is divided into two parts: one for token

scanning, one for parsing. The BNF for token scanning is ambiguous when the scanner encounters a string of characters. According to the BNF, a space may appear as zero repetitions between characters in a name or number. This ambiguity is resolved in the obvious manner: in every case, the largest legal string is chosen as the next token.

BNF For Token Scanning

```
file           = space { token space }
token          = name | number | keyword | specialChar | quotedString
keyword        = "CELL" | "MACRO" | "CONNECTORS" | "COMPONENTS" | "TWIGS" |
                 "CONSTRAINTS" | "TOP" | "LEFT" | "BOTTOM" | "RIGHT" | "ARRAY"
specialChar    = "(" | ")" | ";" | "." | ":" | "[" | "]" |
                  "<" | ">" | "=" | "-" | "+" | " " "
quotedString   = string {string}
string         = " " " {any character except quote} " " "
name           = letter {nameChar}
number         = [ "-" ] digit {digit}
letter         = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
nameChar       = letter | digit | "_"
digit          = "0" | "1" | "2" | ... | "9"
space          = {sepChar} | space "[" commentText "]" space
sepChar        = any character except nameChar, specialChar
commentText    = {commentChar} | commentText "[" commentText "]" commentText
commentChar    = any character except "[" or "]"
```

BNF For Parsing

```
File            = [TechnologyName]  {celldef}
CellDef         = CellHeader {cellDef} CellBody "END" | MacroHeader "END"
CellHeader      = "CELL" name number number point point HeaderExt ConnectorSpec
MacroHeader     = "MACRO" name number number point point
                  CellType  quotedString HeaderExt ConnectorSpec
HeaderExt       = {any token except "CONNECTORS"}
CellBody        = ComponentSpec TwigSpec ConstraintSpec
ConnectorSpec   = "CONNECTORS" {ParamText ":"
                  ConnName point {ConnName point} ";" }
ComponentSpec   = "COMPONENTS" {ComponentType ParamText ":"
                  CompDecl {CompDecl} ";"}
TwigSpec        = "TWIGS" {ColorName ParamText ":" [twigname] "="
                  TwigPrimitive TwigEntry {TwigEntry} ";"}
ConstraintSpec  = "CONSTRAINTS" {ConstraintStmt ";"}
ComponentType   = name | POINT
CompDecl        = CompName [ArraySpec] [orientation] point
ParamText       = {name ParamTok}
ParamTok        = name | number | quotedString |
                  "(" { name | number | quotedString } ")"
ArraySpec       = "ARRAY" nx, y dx dy "X" conns "Y" conns
conns           = { connname connname }
orientation     = ("N" | "M") number number
TwigEntry       = TwigPrimitive | {"(" TwigEntry ")"}
TwigPrimitive   = CompName ["." ConnName] [ArrayIndicator]| point | ConnName
ArrayIndicator  = "." number "." number
ConstraintStmt  = "Y" YPrim OrderOp YPrim {OrderOp YPrim}  |
                  "X" XPrim OrderOp XPrim {OrderOp XPrim}  | ConstrExt
YPrim           = (ConstrPrimitive | "BOTTOM"  | "TOP") [Displacement]
XPrim           = (ConstrPrimitive | "LEFT" | "RIGHT") [Displacement]
ConstrPrimitive = CompName [.ConnName] | ConnName | Number
OrderOp         = "<" | ">" | "="
Displacement    = ( "-" | "+" ) PositiveNumber
ConstrExt       = any token except "END" { any character except ";" }
ColorName       = name
CompName        = name
TwigName        = name
CellType        = name
ConnName        = name
TechnologyName  = name
PositiveNumber  = digit {digit}
point           = number number
```

The file contains a list of cell definitions. Each cell definition begins with either "CELL" or "MACRO" and ends with "END". A "CELL" description is divided into four sections: Header, Component Definition, Twig Definition, and Constraints. Each section starts with a keyword. Cell definitions may be

nested within other cell definitions. A "MACRO" description has only the header section.

All numbers are integers, and are considered to be in units of hundredths of a micron. The header of the cell has a scale factor to be applied to the numbers in the cell. Comments can be included anywhere a space can be put and can be removed in the lexical scan.

Whenever two consecutive quotes are encountered in a quoted string, they are to be interpreted as single quote within the string.

B.3. Semantics

The Sticks file consists of components, interconnect, and constraints on the physical layout. Each file can be prefixed with an optional technology type, for example nMOS or CMOS. The technology type can be useful to select a set of technology-dependent names for components and layers. An example for nMOS is given at the end of this document. The header gives the cell name, scale, abutment box, and connectors. The component definition indicates the type of each kind of component. The twig definition section describes the twigs, their connections, and their paths. The constraints section specifies restrictions on the physical layout.

The points given with the components and twigs, and in constraint displacements give a sample physical arrangement of the components and twigs. These physical locations and distances may be considered mere suggestions by a Sticks processing program.

B.3.1. Cell Definitions

Cell definitions can define either a Cell or a Macro. A Cell has its complete definition included in the Sticks Standard file, while Macro cells are those that are defined elsewhere. Macros may include cells from a library of hand-drawn circuitry, PLAs, ROMs or other cells produced from specialized generators. The precise description of these cells is not included in the Sticks Standard file, but is contained in the external file whose name is listed in the Macro definition. The Macro definition specifies the interface to those cells in a uniform manner. No nested cell definitions, components, twigs, or constraints sections are allowed in Macros.

The Header

A Cell header consists of the cell name, scaling factor, abutment box, optional header extensions, and connectors. A Macro header consists of the name, scaling factor, abutment box, Macro type, external file name, optional header extensions, and connectors.

The scaling factor consists of two numbers: A and B. A specifies the value of lambda in hundredths of a micron and B specifies the value of lambda in Stick file units. For example, A=250, B=4 specifies that lambda=2.5 microns and that one Stick file unit is one-quarter lambda. If units of hundredths of a micron are desired, then all numbers in the cell are scaled by A*number/B. The scaling is not applied to instances of other cells declared in the component list or to cells whose definitions are nested within the current cell.

The abutment box is specified by two points that define a box. The first point

is the lower left corner of the box and the second point is the upper right corner. In the composition of cells this box is treated as the cell outline, with the connectors as fingers on or inside of the box. It should be noted that the abutment box can be different from the minimum bounding box of the cell.

The optional header extension field is useful for adding new or non-standard features to the cell header. The only restriction on this field is that it may not contain the token "CONNECTORS".

The connectors are named locations to which connections to instances of the cell are made. Connectors may be parametrized. For example, the layer of the connector, the side of the cell from which the connector should be attached from outside the cell, or the connector type all could be useful parameters. Examples of connector parameters are again in the nMOS technology definition later.

The Macro type is the general class to which a Macro cell belongs, such as a cell defined as geometry. The external file name specifies as much information as is needed for a Sticks reader to locate a Macro cell for those applications where a Sticks reader must examine the external file on which the complete definition of a Macro exists. As with other quoted strings in the Standard, two consecutive quotes inside the fill name is interpreted as a single quote in the string.

B.3.2. Component Definition

A component may be a Point, a technology-dependent name, or a reference to a cell defined in the current context. The predefined component Point is

simply an interesting location in the design. Twigs may be routed through Points to provide the initial topology of the design. The component Point may be used in several twigs to ensure that all twigs pass through the same physical location.

Technology-dependent names denote transistors, resistors, contacts, and similar features in that technology. The reader should refer to the specification of the technology that he is using for technology-dependent component definitions.

A cell reference, denotes an instantiation of the cell, the placement of the cell's contents at the position and in the orientation specified by the transformation on the component.

Each component definition includes a position and an optional orientation, which includes mirroring and rotation information. This orientation, like the physical locations on components, is mere suggestion, and may be altered by a Sticks processing program. The orientation consists of a one-letter indication of the mirroring of the coordinate system. A "N"ormal coordinate system has the +y axis counterclockwise from the +x axis. A "M"irrored coordinate system has the component mirrored about the (1,1) vector (i.e. the x and y values are swapped). The two numbers following the mirroring key is the rotation key. These two numbers give the x-y coordinate of the direction to which to rotate the +x axis. When transforming a component, mirroring is done first, followed by rotation then translation.

Array Specification

The array specification requires the replication count in x and y (nx and ny),

which must at least 1. Also, the spacing of the array must be given in the same units as other measurements in the cell. As with other positions and sizes, the array spacing may be modified by processing programs. The user must also specify the interior connections in the array. All interior connections must be the same. These connections imply twigs between the connectors on cells. Connections are made either horizontally ("X") or vertically ("Y"). The first connector name in the interior connection specification refers to the i^{th} element of the array, the second connector names refers to the $i+1^{st}$ element.

Positioning and orientation of an array are relative to the origin on the (1,1) element of the array. Specific connectors on array elements can be referenced in twigs by referring to the array index.

B.3.3. Twig Definition

A Sticks twig is a connected path with a given layer and set of parameters (for example, line width). The twig may or may not be a true electrical node, as it may run into a contact component that may make contact to the same layer or other layers. It may intersect other twigs. The electrical interaction at the crossing of twigs is not specified, and the Sticks Standard does not restrict the crossing in any way. Such a construct might violate design rules in a particular technology, and could be checked. Twig attributes cannot be changed as they are specified only once for the twig.

A twig optionally has data following the layer name to allow specification of technology-dependent information. The legal layer names and the parameters for a twig must be defined as part of the technology-dependent

parts of Sticks for a particular technology.

The twig definition describes the path taken by the twig. A path consists of two or more points and can have branches. Mere connection can be easily represented for those applications where paths are unnecessary or redundant. A branch in a twig is represented by a parenthesized point list. The TwigEntry in parentheses branches off the point given immediately before the parentheses. Therefore no twig may start with a branch point.

A specific connector on a component is referenced by the component name followed by a dot and the connector name. If a twig is routed to a component that is referenced by its component name only, then the connection point is assumed to be the origin of that component. Unnamed points may be specified directly in the twig definition using a coordinate pair. Two unnamed points at the same coordinate are assumed to be separate points and their identical positions mere co-incidence.

A connection to an array element is specified by following the connector name with a dot then the x-index, another dot and the y-index. Omitting the array element number causes a bus connection to be made to all array elements. If the twig connects two differently-sized arrays, array element 1 connects to 1, 2 to 2 and so forth until one array has no more elements to be connected. Bus twigs should be used with caution, since all elements in the bus must be arrays.

The twig may be named if desired. The name is not used in the Sticks Standard, but may be useful later in viewing the Sticks or assembling the Sticks data into a form useful for simulation. There is no guarantee by the Sticks Standard that twigs with the same name all belong to the same

electrical node.

B.3.4. Constraints

A constraint is a restriction on the physical location of a component or a connector. Constraints can be considered instructions to a Sticks processing program to limit the deformation performed by that program.

There are three constraint operators for ordering: <, >, =. They are the familiar relational operators and force their ordering.

There are four keywords in the Constraint section: LEFT, RIGHT, TOP, BOTTOM. They refer to the four edges of the abutment box of the cell. It is implied that LEFT < everything < RIGHT and BOTTOM < everything < TOP. A point can be constrained to be equal to one of the keyword values, in which case it will always be positioned at the appropriate edge of the cell. The keywords need not be used in the description, but their values represent the physical limits of the cell and as such may be very helpful when setting connections to the outside world.

An optional displacement can be added to a constraint. This displacement sets a limit to deformation offset from a component. In addition, an orderPrimitive may be a number that restricts the legal physical locations of a component. These numbers are in the same units as the others in the cell, and like those others are mere suggestion and are expected to be changed by Sticks processing programs.

Constraints apply in either the horizontal (x position of components) or vertical (y position of components) direction. They are applied in pairs, so

"a1 < a2 < a3;" is just shorthand for "a1 < a2; a2 < a3;". The constraint section also has provisions for non-standard constraints to be specified with the constraint extension that allows extensions to be intermixed with standard constraints. A constraint extension cannot start with the token "END" and it is terminated with a ";".

Circular constraints can be built, and the reader of the Sticks Standard file is advised to beware of them.

B.3.5. The Definition Hierarchy

Cell definitions may be nested within other cell definitions to arbitrary depth. Any cell can be used as a component within any other cell subject to the following constraints that limit the scope of each cell definition: A cell definition must be complete (to its "END" statement) before it can be used as a component. There may no self-references or forward-references. All cells defined within the definition of any cell at any level are considered local to that cell and may not be used as components by any cell defined outside that cell. If two cells have the same name and both are accessible by some cell, the cell most-recently defined shall be used when reference is made to that name. The following example demonstrates the cell nesting.

Cell definitions as they occur in the file.	Cells that can be used within the given body.
CELL A ...	
CELL B ...	
[BODY B]	none
[BODY A]	B
CELL C ...	
CELL A ...	
CELL D ...	
[BODY D]	first A
[BODY A]	D, first A
CELL E ...	
[BODY E]	second A
[BODY C]	E, second A

[BODY x] = entire definition of cell x except for the Cell Header. Indentation is used to emphasize the hierarchy.

B.3.6. Technology-Dependent Parts

There are pieces of a Sticks specification that are not predefined by "pure" Sticks. This allows the framework of the Sticks Standard to be used for more than one integrated circuit technology. For a given technology, the component names and twig layers must be named, their parameter syntax and semantics must be given, and the geometrical representation of the components and twigs must be specified if geometric manipulation of the Sticks is to be done. The names may be made deliberately unique within an integrated circuit technology as well as across technologies to avoid confusion. Parameters to components may or may not have precise geometrical or electrical meaning.

Component, twig, and connector parameters are similar in their form. All are specified as a series of pairs where the first token of each pair is a technology-dependent keyword and the second is the value associated with the keyword. A value that consists of more than one token must be

surrounded by parentheses. This allows the equivalent of extensions in the parameter lists since the procedure for getting to the next parameter is always known.

Component parameters give transistor dimensions or similar information. Twig parameters identify wiring layer widths or current capacity. Connector parameters give types and layer information. All technology-dependent specifications should become appendices to this document. The technology-dependent specification for nMOS is given below.

B.3.7. Parametrization of Instances

In most applications, the position of components in an instance must be different than the positions specified in the definition. The parameter passing mechanism for components is used to pass a list of the desired transformation of the components for the individual instance. If more transformations are given than there are components in the instance, then the last ones are ignored. If fewer coordinate pairs are specified than there are components in the cell, then the remaining components are placed using the sample positions in the cell. The form of these parameters is given in the at the end of this document.

B.4. nMOS Sticks Standard Components

This is a summary of the definitions of the nMOS Sticks in [Kahle 1981]. These definitions have been widely used and have been found adequate for a wide variety of applications. Not all users have implemented all the parameters, though.

B.4.1. Component Definition

The nMOS components are shown in figure B.1. They are summarized below:

NENH and **NDEP**

NENH is an enhancement mode transistor, NDEP is a depletion mode transistor. The origin of the transistor is at the center of the gate area. In both, the gate is horizontal, the drain is at the top, the source is at the bottom. The connectors on the to transistor are:

G1	the left side gate connector.
G2	the right side gate connector.
DRAIN	the transistor drain connector.
SOURCE	the transistor source connector.

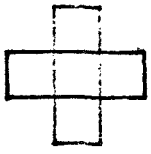
The parameters to the enhancement transistor are given in the table below. The value in brackets is the default value.

W <num>	width of the transistor [2].
L <num>	length of the transistor [2].
G1 <point>	the location of the G1 gate connector [(-2,0)].
G2 <point>	the location of the G2 gate connector [(2,0)].
DRAIN <point>	the location of the drain connector [(0,2)].
SOURCE <point>	the location of the source connector [(0,-2)].

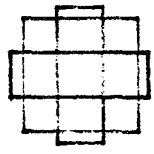
A path can be given for either the gate or the drain. The reader is referred to [Kahle 1981] for details.

NRES and **NBRES**

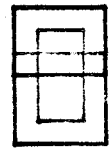
These are pullup resistor type depletion mode transistors with their source shorted to the gate. NRES uses a butting contact to make the connection, NBRES uses a buried contact. In both, the origin is the center of the contact.



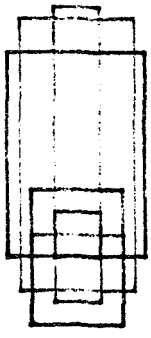
NENH



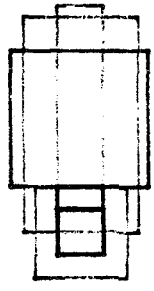
NDEP



NBUT



NRES



NBRES

NBUR

NPM

NDM

NCON

Figure B.1. nMOS Sticks Standard Components.

Their connectors are:

DRAIN	the transistor drain connector.
PSOURCE	the transistor source connector on poly.
DSOURCE	the transistor source connector on diffusion.
MSOURCE	the transistor source connector on metal (NRES only).
SOURCE	the transistor source connector on unknown layer.

The parameters to the resistors are given in the table below. The value in brackets is the default value.

W <num>	width of the transistor [2].
L <num>	length of the transistor [6].
DRAIN <point>	the location of the drain connector [(0,9)].
PSOURCE <point>	the location of the poly source connector [(0,0) for NBRES, (0,2) for NRES].
DSOURCE <point>	the location of the diffusion source connector [(0,0) for NBRES, (0,-2) for NRES].
MSOURCE <point>	the location of the metal source connector [(0,0) NRES only].

NBUT

This is a butting contact. The polysilicon part is above the diffusion part. Its connectors are:

P	the poly connection.
D	the diffusion connection.
M	the metal connection.

The parameters to the butting contact are given in the table below. The value in brackets is the default value.

W <num>	width of the contact area [4].
P <point>	the location of the poly connector [(0,2)].
D <point>	the location of the diffusion connector [(0,-2)].
M <point>	the location of the metal connector [(0,0)].

NBUR

NBUR is a buried contact. It is symmetrical, and has two connectors that default to its origin:

P	the poly connection.
D	the diffusion connection.

The parameters to the buried contact are given in the table below. The value in brackets is the default value.

W <num>	width of the buried contact area [4].
L <num>	length of the buried contact area [4].
P <point>	the location of the poly connector [(0,0)].
D <point>	the location of the diffusion connector [(0,0)].

NPM, **NDM** and **NCON**

These are simple contacts to metal. NPM is a polysilicon to metal contact, NDM is a diffusion to metal contact, and NCON is an uncommitted contact. They are all symmetrical, and have no connectors, so all connections must contact the center (0,0).

The parameters to these contacts are given in the table below. The value in brackets is the default value.

W <num> width of the contact cut area [2].
L <num> length of the contact cut area [2].

CONNECTOR

The final component is the connector, which has no geometrical representation. However, it can have one parameter, a type, indicated by the key letter T. The legal types are: INPUT, OUTPUT, IO, POWER, GROUND, CLOCK and BUS.

B.4.2. Twig Definition

The nMOS twig can have the following layers: POLY, DIFFUSION or METAL.

B.4.3. Instance Parameters

It is desirable in some design systems to pass a value that is a position to a component inside an instance in the current cell. This allows the same cell definition to be used for several positions of the components, several "compactions". The parameter specification to do this is:

C (<compname> [<orientation>] <point>)

Where <compname> is the name of the component in the cell definition for the instance, <orientation> is a Sticks orientation specification, and <point> is the location in the instance's coordinates where the component should be placed. If the orientation is omitted, the orientation that the component in question already has is used.

B.5. Example of the Sticks Standard in Action

The following is a shift register cell in the Sticks Standard form using nMOS components.

```
NMOS
CELL srcell 250 4 [ lambda = 2.50 microns ] -48 -59 48 59
 CONNECTORS
  T GROUND: gndl -48 -45  gndr 48 -45 ;
  T INPUT: in -48 -29 ;
  T POWER: vddl -48 45  vddr 48 45 ;
  T OUTPUT: out 48 -29 ;
  T CLOCK: clktop 8 59  clkbot 8 -59 ;
 COMPONENTS
  NENH W 16 L 8: pd -20 -29 ;
  NENH W 8 L 8: ps N 0 -1 8 -7 ;
  NRES W 8 L 32: pu -20 1 ;
  NBUT: but N -1 0 28 -15 ;
  NDM: N1 -20 -45 ;
  NDM: N3 -20 45 ;
 TWIGS
  POLY:= clkbot 8,-43 ps.G1 clktop;
  METAL:= gndl N1 gndr;
  DIFFUSION:= N1 pd.SOURCE;
  POLY:= in pd.G1;
  DIFFUSION:= pd.DRAIN pu.DSOURCE ps.SOURCE;
  POLY:= 28,-29 (out) but.P;
  DIFFUSION:= pu.DRAIN N3;
  DIFFUSION:= ps.DRAIN but.D;
  METAL:= vddl N3 vddr;
 CONSTRAINTS
  X clkbot = clktop;
  Y gndl = gndr;
  Y in = out;
  Y vddl = vddr;
  in.Y=out.Y;
END

CELL sr 250 4  -50 -59 350 59
 CONNECTORS
  T POWER: PWRIN -50,45 PWROUT 350,45;
  T GROUND: GNDIN -50,-45 GNDOUT 350,-45;
  T INPUT: INPUT -50,-19;
  T OUTPUT: OUTPUT 350,-19;
  T CLOCK: CLKTOP ARRAY 4,1 100 0 6,59;
  T CLOCK: CLKBOTTOM ARRAY 4,1 100 0 6,-59;
 COMPONENTS
  SRCELL: sreg ARRAY 4,1 100,0
    X vddr vddl gndr gndl out in
    Y
    0,0;
 TWIGS
  POLY:BCLOCKS = CLKBOTTOM sreg.clkbot;
  POLY:TCLOCKS = CLKTOP sreg.clktop;
  METAL:= PWRIN sreg.vddl.1.1;
  METAL:= PWROUT sreg.vddr.4.1;
  METAL:= GNDIN sreg.gndl.1.1;
  METAL:= GNDOUT sreg.gndr.4.1;
```

```
  POLY:= INPUT sreg.in.1.1;
  POLY:= OUTPUT sreg.out.4.1;
 CONSTRAINTS
END
```

B.6. References

[Kahle 1981] C. Kahle and R. Mosteller, "NMOS Sticks Standard Components", Computer Science Department Display File #4300. California Institute of Technology.

[Wirth 1977] N. Wirth, "What Can We Do About the Unnecessary Diversity of Notations for Syntactic Definitions?", Communications of the ACM, 11/77

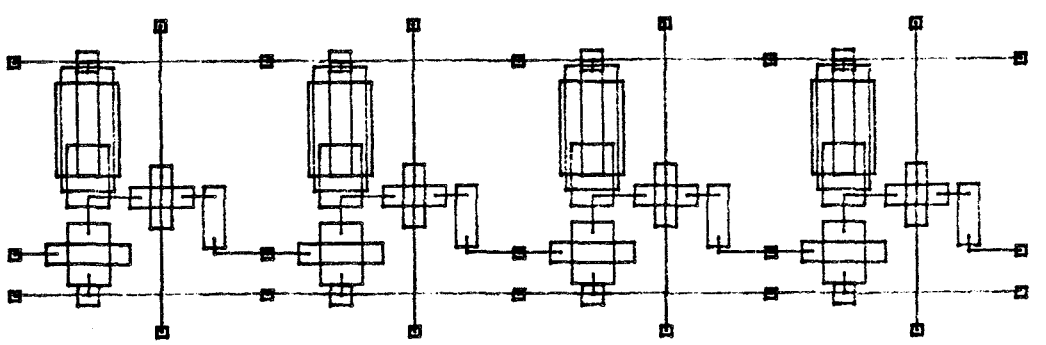


Figure B.2. The Shift Register Example