# Software Framework for CMOS Circuit Optimization Using Convex Optimization

Alvin Cheung
Department of Electrical Engineering
akcheung@cs.stanford.edu

## I. Introduction

A modern digital circuit consists of gates that perform different logical functions. Each logic gate, in turn, is constructed from a number of MOS transistors. An important part of the design process involves the sizing of the circuit's transistors, as sizing has direct impact on the performance. Since sizing is a critical problem in circuit design, many tools have been constructed to solve this problem, from simple back of the envelope calculations to sophisticated non-linear optimizers that run detailed circuit simulations. These tools vary in terms of the complexity of circuits they can analyze and the amount of time required to generate the results.

To build an efficient optimzer, we use generalized geometric programming (a class of convex optimization) techniques in our framework, which consists of three distinct modules: a netlist parser, a circuit analyzer, and a convex problem solver. The netlist parser takes in netlists in a modified SPICE format and transforms them into a hierarchical graph structure, with nodes in the graph representing circuit elements and connection points, and arcs in the graph representing connections in the design (details of the graph structure will be explained in Section II). This intermediate representation is stored internally in memory. Given this graph, the circuit analyzer generates a set of delay equations between the inputs and output of each subcircuit based on the various transistor delay models provided by the user. These equations are fed into the convex problem solver using the Mosek [7] software package to find an optimal solution, and the results are recorded in a file that are annotated back into the input SPICE netlist.

This thesis is primarily concerned with the design and implementation of the first two modules. These modules are grouped into a software package called diogen and are used to transform a circuit into a circuit independent, abstract mathematical problem that can be solved using a

mathematical solver. These modules handle netlist syntax and circuit modeling and allow the user to incorporate different circuit delay models into the framework. The parser uses standard software parser generation tools and a CAD programmer can extend the grammar to analyze circuits described in a different syntax. As long as the intermediate graph representation of the circuit remains the same, the user can still use our analyzer framework to perform optimization. Figure 1 is a graphical representation of the functionalities and software framework in diogen.

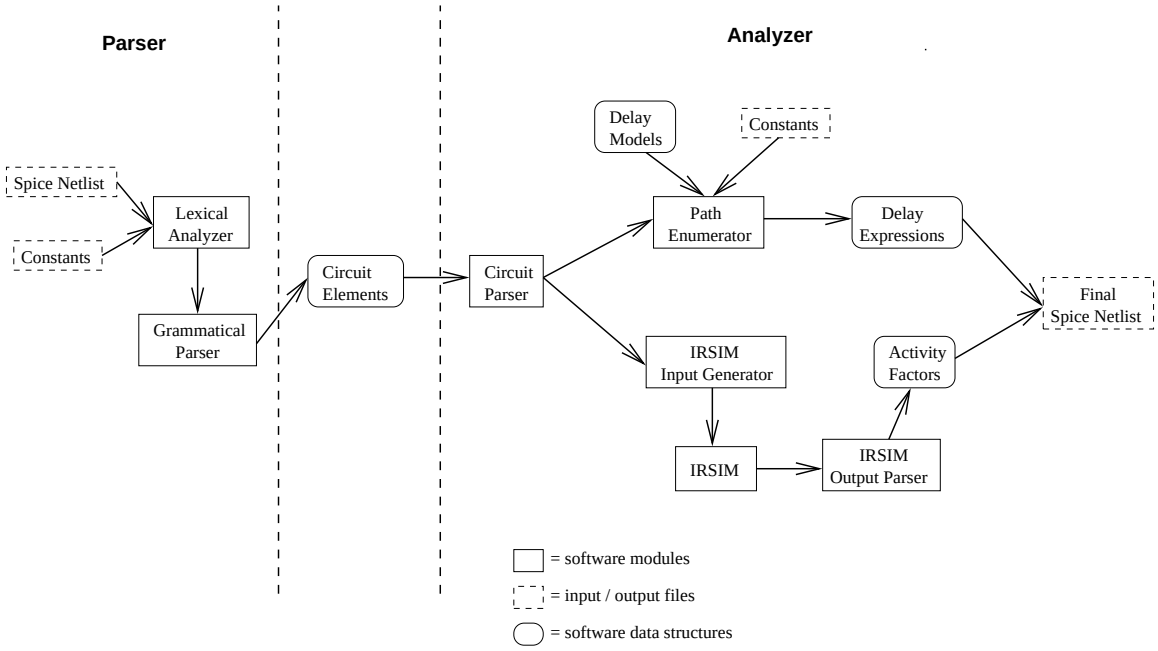


Fig. 1. System Overview

### *A. Previous Work*

One of the earliest circuit optimizer is TILOS [3], which models transistors by equivalent RC circuits and perform static tuning. In TILOS, all transistors start with their minimum widths. TILOS then identify the critical paths in the design, and incrementally increase the width of the transistors to which the critical path is the most sensitive to until the lowest critical path delay is found. More recent approaches include EinsTuner from IBM [1], which is a circuit analyzer built from IBM's proprietary circuit parser and optimizer. EinsTuner performs static analysis by formulating delay equations of the design based on EinsTuner's delay models and subsequently using an internal mathematical solver to obtain the results. While this approach requires less

time than the TILOS iterative approach, the usability of the optimizer is limited by the classes of circuits that it can analyze due to the availability of delay models, and the optimization results are also dependent on the accuracy of the delay models that the user is not able to alter. Pathmill [9], on the other hand, is a static timing analysis tool from Synopsys that tries to determine the critical path in a given netlist by simulating the design and recording the timing results. It can analyze a large number of different circuit classes, although the simulations can take a long time to complete.

The goal of our work is to build a circuit optimizer that does not require lengthy circuit simulations while being able to handle a variety of circuit classes. As a result, we design a software architecture that allows the user to provide different delay models for different classes of circuits. We then use the models provided by the user to transform the optimization problem into a mathematical format for the solver.

### *B. Thesis Layout*

Section II presents the parsing methodology used in diogen, along with the circuit elements that are currently supported. In Section III, the formulation of delay equations given the circuit delay models will be discussed together with issues that arise when processing different classes of circuits. We then discuss activity factor generation methodology for circuit power optimization in Section IV and conclude with a summary and future work in Section V.

## II. NETLIST PARSING

In a typical SPICE deck, a design consists of connected modules called subcircuits, and each subcircuit is made up of circuit elements and circuit nodes. Circuit elements are physical devices such as transistors, and circuit nodes represent connection points between two circuit elements (ground and $V_{dd}$ are also represented as circuit nodes). Thus, two physically connected devices are represented in SPICE as two circuit elements connected via a circuit node, and a circuit node can be connected to any number and type of circuit element.

As the frontend of the optimization framework, the netlist parser is responsible for transforming the input netlist into an internal representation. This internal representation is based on the SPICE

representation and is in the form of a two-level hierarchical graph.[1] The top level consists of subcircuit instance nodes, which represents a SPICE subcircuit instance and is assumed to be a channel connected component (defined in Section II-B). Each subcircuit instance node contains a pointer to a subcircuit node of its type, and each subcircuit node in turn consists of a lower level graph that is made up of two types of nodes, circuit element nodes, which represent physical devices inside the subcircuit, and connection nodes, which are counterparts to circuit nodes in SPICE. Similar to SPICE, in the internal representation two circuit elements are also connected via a connection node, and each connection node stores pointers to the circuit element nodes that are connected to itself. Ground, $V_{dd}$, subcircuit input, subcircuit output are all represented as special types of connection nodes in the internal representation.

### *A. Parser Inputs*

In the current version, diogen parses two files from the user. These files are:

- A global parameters file. This file consists of two portions. The first part contains constants such as values of $V_{dd}$, $V_{th}$, and per lambda values of parasitic capacitance. These constants are used in generating the delay equations of diogen. The second half of the parameters file contains the delay models for different transistor stacks. Each delay model is written as a mathematical function of the transistor widths in the given stack. In the current version transistor stacks up to four transistors are supported, along with both static and statistical versions of each model. Delay models are further explained in Section III-A.
- A SPICE input file. Like a regular SPICE deck, this file consists of subcircuit definitions and connectivity information in SPICE syntax. Each subcircuit is assumed to be a channel connected component (CCC) as defined in the next section. In addition, the SPICE input file also contains a global constraint section that allows the user to provide constraints such as width constraints of individual transistors in a given subcircuit, or overall area / power constraints on the entire design. The final portion of the input file consists of an optimization target section that specifies which type of optimization (area / power) is to be performed.

[1]In SPICE there is no limit to the nesting of subcircuits. Each subcircuit can contain further subcircuits. However, for the ease of circuit analyzing in our framework, we use a script to flatten all subcircuits such that each subcircuit contains only circuit elements and no further subcircuits, and the modified netlist is parsed. The flattening functionality can be integrated into the parser in the future so that no pre-processing step is necessary.

### *B. Channel Connected Component*

A channel connected component (CCC) is defined to be a set of transistors connected to at least one other transistor in the same set through its drain or source terminals. In our formulation, the connection point between two transistors is called a circuit node (as in SPICE). As mentioned, the circuit parser assumes that each subcircuit defined in the input file is a channel connected component, thus implying that the user has pre-partitioned the entire design into channel connected components beforehand. While this might present additional constraint to the user, this requirement greatly simplifies the task of generating the output delay equations. Because a transistor stack is also a channel connected component, the generation of output delay equations reduces to mapping the appropriate transistor stack delay models to each channel connected component and combining the expressions together to form the final output (described further in Section III). If each subcircuit in the input file is allowed to contain arbitrary structures, the generation task will be much more difficult. In that case we will need to pre-process each subcircuit definition to ensure that each one is a channel connected component, and divide the subcircuit into connected components if necessary and parse on the modified netlist instead. In general, since most typical CMOS gates such as inverters, NAND, or NOR gates are already channel connected components, the user only needs to separate each of the logic gates in an individual subcircuit and will be able to use our framework.

### *C. Choices for Internal Representation*

There are a number of choices for the internal representation of SPICE netlist besides our current implementation. Our decision was based on the following observations:

- Ease of implementation. Since our optimization framework takes in SPICE netlist as inputs, it is natural to use an internal representation that resembles SPICE that is easy to implement.
- Circuit analysis. Since the delay expression generation process is done on a per subcircuit basis, using a hierarchical internal representation, as opposed to a flattened, non-hierarchical representation, makes the analysis step much easier to implement.
- Ease of parsing. There are three major reasons for the use of connection nodes instead of directly connecting two adjacent devices together. First, SPICE uses the same format to represent connectivity. Secondly, parasitic capacitance (discussed in Section III-B) is formulated as the capacitance between circuit elements in our framework, thus with con-

nection nodes we can simply iterate through each connection node and perform parasitic capacitance calculation. Otherwise, we will need to identify all connection points between circuit elements at a later point when performing parasitic capacitance calculation. Thirdly, if we did not have connection nodes then we will need to use a two-pass process when parsing each subcircuit in order to generate connectivity information of each circuit element. In the first pass, we will need to create a hashtable that maintains pairs in the form of {SPICE circuit node name, circuit element objects that are connected to the node}, and temporarily store as connectivity information the SPICE circuit node name that each circuit element is connected to. In the second pass, we will need to iterate through each circuit element in the subcircuit and replace the SPICE circuit node name with the circuit element objects in the hashtable. However, if we maintain connection nodes, and have each circuit element stores pointers to the connection nodes, then we simply need to update the connection node object for connectivity information as we progress through each subcircuit, and we only need to iterate through each subcircuit once.

### *D. Parsing Process*

The parser in diogen is built upon the GNU flex and bison suite [4]. Flex and bison are automatic lexical analyzer and parser generator tools based on grammar files inputted by the user. Given the input files, the parser in diogen first parses through the global parameters file and store each of the parameters and delay models as {name, value} pairs in a hashtable for the ease of retrieval later during delay expression generation.

In the second step, the parser goes through the spice netlist and examines each of the subcircuit definitions. For each subcircuit definition, the parser creates a subcircuit node and records the inputs and output of the subcircuit. In order to maintain connectivity information among the circuit elements within the subcircuit, a temporary connectivity hashtable is constructed during the parsing of each subcircuit. Each entry in the hashtable is in the form of {circuit node name from SPICE netlist, pointer to the corresponding connection node object in internal representation}.

The parser then goes into each subcircuit and performs the following procedure on each circuit element definition:

1) Create the circuit element object of the appropriate type.

2) For each connection point of the circuit element, look up in the connectivity hashtable to see whether the connection node object already exists.
3) If the connection node object already exists, then add the current circuit element object to its connection list. If the connection node object does not exist, then create a new connection node object, add the current circuit element object to its connection list, and finally add the connection node object to the connectivity hashtable.

Note that in this representation each subcircuit can have its own ground and $V_{dd}$ objects and nodes of the same name that appear in different subcircuits are considered as different nodes. However, since both parsing and delay expression generation are performed on a per subcircuit basis, having different ground or $V_{dd}$ connection nodes in each subcircuit does not create any problems.

After each subcircuit definition is parsed, the parser processes the global connectivity information in the SPICE netlist, which describes the instantiation of each subcircuit type in the design and the connectivity among the subcircuit instantiations. For each subcircuit instantiation, we create a subcircuit instance node and add in it a pointer to the corresponding subcircuit node object that was previously generated. A similar hashtable is constructed to store the connection node objects among the subcircuit instances and maintain connectivity information.

In the final parsing step, the primary inputs section, which is our newly added section in the SPICE netlist, is parsed with the names of all primary inputs stored in an array. Primary inputs are circuit nodes that are not driven by any other node in the same design. These inputs are used later during activity factor generation to create input vectors to IRSIM and is discussed further in Section IV.

At the end of parsing, the parser passes the circuit instance nodes, the subcircuit nodes, and the primary input list to the circuit analyzer, which in turn processes these information to generate delay expressions.

## III. Delay Expressions Generation

After the parser creates an internal representation of the input netlist, the circuit analyzer processes the internal representation to generate delay expressions. For each channel connected

subcircuit in the circuit, a delay expression is generated between each input and output.[2] These delay expressions represent the internal delay of each subcircuit and are stored in a file that is passed to the convex problem solver. Similar to the parser in diogen, the convex problem solver constructs a graph representation of the design and formulates delay equations for each node, with each node being a subcircuit instantiation in the design. In the solver, delay equations are defined recursively in the form of

$$\begin{aligned}&\text{delay at the output of subcircuit } n = \\ &max(\text{each input arrival time in subcircuit } n + \\ &\text{internal delay through subcircuit } n \text{ from that input to output}) \end{aligned} \tag{1}$$

In the formulation each arrival time is the delay equation from the output of another subcircuit instance node, and internal delay through a subcircuit is defined to be the maximum of all the delay expressions generated by the diogen circuit analyzer from a given input to the output.[3] After the delay equations for all circuit nodes are generated, they are given to the optimizer to solve according to the constraints that were given by the user. In the following, we explain the delay expression generation process in each subcircuit node and the current delay models that are used in the framework.

### *A. Delay Models*

In circuit design, gate delay is often characterized as the following after the Elmore delay model [2]:

$$\text{delay} = \frac{CV_{dd}}{2I} \tag{2}$$

where $C$ represents the total load and parasitic capacitance driven by the gate, and $I$ is the current through the gate, with the term $V_{dd}/I$ being known as the effective resistance ($R_{eff}$) of the gate. The Elmore model gives a method to calculate delay in a $RC$ network. In this model, we consider $RC$ networks that have a single input node, with all capacitors being between a node and ground, and the network does not contain any resistive loops. Because of these

[2]The current system supports only subcircuits with one output, extensions to multiple output systems is relatively simple from the parsing perspective but will involve revising most of the current delay models from the circuit modeling standpoint.

[3]If an input node to a subcircuit instance is a primary input its arrival time is taken to be 0.

properties, there exists a unique resistive path from the source to each of the nodes. We define path resistance, $R_{ii}$, to be the total resistance along the resistive path to node $i$. We further define shared path resistance, $R_{ik}$, to be the resistance of the path to ground that is common to both nodes $i$ and $k$ from the source. Assuming that each of the capacitors in the network is initially discharged to ground, the Elmore delay model states that if a step input is applied at the source at time zero, then the delay for node $i$ is given by:

$$d_i = \sum_{k=1}^{n} R_{ik} C_k \tag{3}$$

As an example, consider the circuit in Fig. 2. The Elmore delay for node $i$ in the network is:

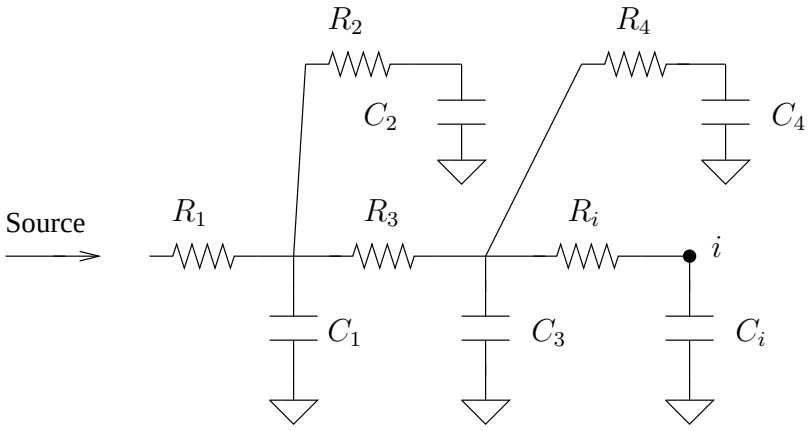


Fig. 2. Elmore Network Example

$$d_i = R_1C_1 + R_1C_2 + (R_1 + R_3)C_3 + (R_1 + R_3)C_4 + (R_1 + R_3 + R_i)C_i \tag{4}$$

In our framework we model each transistor as a resistor, and we use the Elmore delay model to calculate delay. For single transistor stacks, we can directly compute $R_{eff}$ given the size of the transistor. For multiple transistor stacks, one would expect the effective resistance to be the sum of individual transistor's effective resistance. However, since the transistor is fundamentally a non-linear resistor, this approximation is not entirely correct. Thus, we develop models for transistor stacks of different sizes based on the velocity saturated model for transistors [11], and we continue to formulate the transistor circuit as a $RC$ network for delay calculation.

We organize our delay models in the framework by two means: the type of transistor stack it represents, and the nature of the model. In the current implementation transistor stacks of up

to four transistors are supported, and models with transmission gates within the stack is also allowed.[4] For transmission gates, the NMOS-PMOS pair is modeled as a 'combined' transistor with drive strength equals to the weighted sum of each transistor's drive strength. The weight given to the two transistors depends on whether the transistor stack is a pulldown (driving the output from $V_{dd}$ to ground) or pullup (driving the output from ground to $V_{dd}$) one. In the case of a pulldown stack, the transmission gate is modeled as a NMOS transistor with normal drive strength combined with a portion of the PMOS transistor's drive strength, and vice versa for the pullup stack.

In our current system, statistical models for transistor stacks are also included. Currently the models incorporate statistical variations in delay due to variations during the fabrication process. These variations are captured in the model by considering the various device parameters as random variables and generating the variance of the delay as a function of those variables. We use Pelgrom's formulation [10] to model the variance in delay due to the fabrication variations. The basic idea behind the model is that the variations is inversely related to the square-root of the device size and that these variations have very little correlations with other gates in the design. We use the same delay generation process for both deterministic and statistical models as described below.

### *B. Parasitic Capacitance Calculation*

From the discussion above, we know that each circuit delay model represents the effective resistance for the given transistor stack in the $RC$ expression for delay. Meanwhile, the capacitance expression, which represents the amount of charge that the given input needs to charge up or discharge in order for the output to switch, is divided into load and parasitic capacitance. In our framework, we consider load capacitance to be capacitance that a node needs to switch that is outside the subcircuit (channel connected component) that the node resides, and parasitic capacitance to be capacitance that a node needs to switch that is internal to the node's subcircuit. Obviously, only output nodes of each subcircuit will have non-zero load capacitance. In our current framework, load capacitance is calculated by examining the global connectivity information of each subcircuit instance and is performed by the solver. Meanwhile,

[4]In our framework a transmission gate is currently defined as a pair of NMOS and PMOS transistors that has their source and drain connections connected together.

with the connectivity information stored in the connection node of each subcircuit during parsing, parasitic capacitance is calculated in the diogen circuit analyzer for each connection node as needed during delay expression generation using a two step process. In the first step, we identify for each connection node the circuit elements that are connected to that connection node. As an example, consider the circuit fragment in Fig. 3. For the node labeled 'out,' NMOS1, NMOS3,

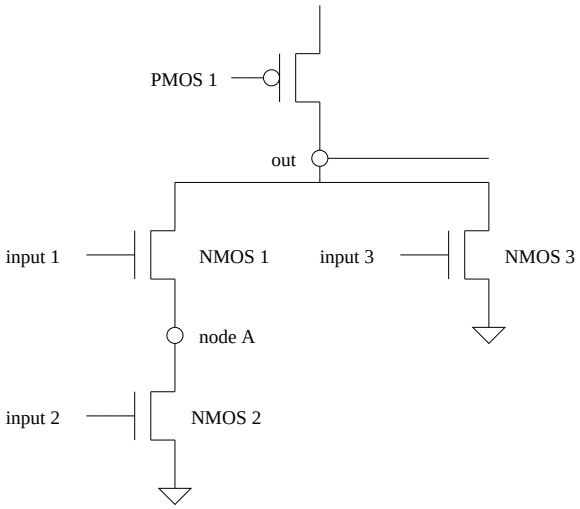


Fig. 3. Parasitic Capacitance Example

and PMOS1 will contribute to parasitic capacitance. Meanwhile, for node A, only NMOS1 and NMOS2 are considered for parasitic capacitance.

After this step, for each transistor that is connected to a connection node, we write the equation

$$\begin{aligned}&\text{parasitic cap. contributed by a transistor at a connection node}\\ &= C_{edge} + C_{gate} + (C_{ov} + C_{width})W \end{aligned} \tag{5}$$

where $C_{edge}$ represents the capacitance of the edge of the diffusion region between the gate to the source / drain, $C_{gate}$ is the capacitance between the gate and the channel, $C_{ov}$ is the gate to source / drain overlap capacitance, $C_{width}$ is the diffusion capacitance of the diffusion contacts, and $W$ is the width of the contributing transistor. The total parasitic capacitance of a connection node is the sum of all individual parasitic capacitance expression as given in Eq. 5.

Note that this is a very crude estimation of parasitic capacitance. For example, we assume that contacts between two adjacent transistors are not shared. Also, $C_{gate}$ of a node, which depends

on the transistors that have their gate terminals connected to that node, is calculated in the solver at a later point as the solver maintains connectivity information among the subcircuit instances.

### *C. Types of Delay Expressions*

As mentioned, delay expressions, also known as dio expressions, are generated between each input and output of each subcircuit. Each expression has the overall form of:

$$\begin{aligned} \text{delay through a subcircuit} &= \sum R_{eff} C \\ &= \sum (\text{transistor stack model})(\text{capacitance driven by the model}) \end{aligned} \tag{6}$$

In our framework, we assume that each subcircuit can be divided into two halves: one that is responsible for driving the output from $V_{dd}$ to ground (pulldown), and one that is responsible for driving the output from ground to $V_{dd}$ (pullup). We further assume that each output node is either at $V_{dd}$ or ground at all times, and thus only one of the two halves in the subcircuit is turned on at any time. With this consideration, we divide delay expressions into the following four categories based on the values of the gate input of each transistor and the output:

- Gate input rising from ground to $V_{dd}$ that causes the output to fall from $V_{dd}$ to ground (rise $\rightarrow$ fall)
- Gate input falling from $V_{dd}$ to ground that causes the output to rise from ground to $V_{dd}$ (fall $\rightarrow$ rise)
- Gate input rising from ground to $V_{dd}$ that causes the output to rise from ground to $V_{dd}$ (rise $\rightarrow$ rise)
- Gate input falling from $V_{dd}$ to ground that causes the output to fall from $V_{dd}$ to ground (fall $\rightarrow$ fall)

The first two delay expression types correspond to the normal operations of NMOS and PMOS transistor stacks respectively. The last two expression types can arise in stacks with transmission gates. Consider the situation in figure 4, when input 3 falls from $V_{dd}$ to ground, the output will also falls to ground as well (assuming that inputs 1 and 2 are driven to $V_{dd}$). Thus the delay expression that is generated for input 3 will be of the fall $\rightarrow$ fall type. A similar situation for rise $\rightarrow$ rise expression type can arise in the NMOS transistor in a transmission gate that belongs to a pullup stack.

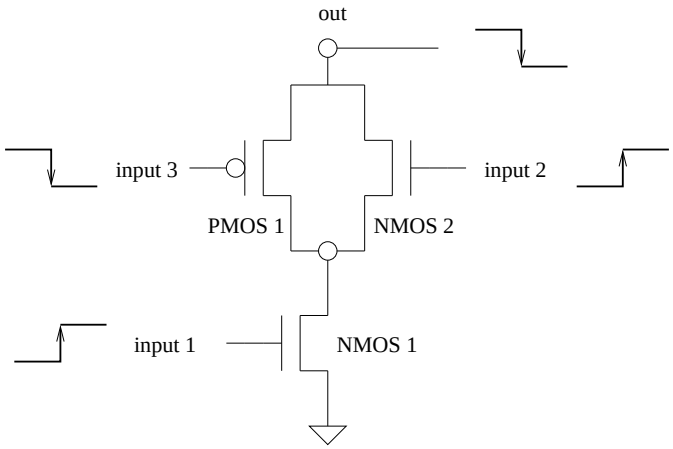


Fig. 4. fall → fall delay expression example

### *D. Software Methodology for Delay Expression Generation*

Given the effective resistance and capacitance information calculated in previous steps, we are now ready to generate delay expressions. As discussed earlier, delay expressions are generated for each path between each input and output of the subcircuit. Here we define a path as a ordered sequence of circuit element and connection nodes, with each node in the path connected to the nodes immediately before and after it. Given our internal representation of the netlist, a complete path will contain alternating circuit element and connection nodes. If there exists only one path from a given input to the output, generating delay expressions is a straightforward process as explained below. However, if multiple paths exists, then we will need to combine the paths together. As an example, consider the circuit in figure 5. Since there are two paths from input 1 to the output node, we need to consider the possible status of inputs 2 and 3 when input 1 is driven to $V_{dd}$. In our formulation we always assume the pessimistic case and consider that only one of input 2 or 3 is at $V_{dd}$ when input 1 becomes high, and we then consider the overall delay from input 1 to the output as the maximum of the two possible delay expressions through each path. Note that while the maximum function represents a pessimistic estimation of delay, it also preserves the convexity of the overall expression. The situation is handled in a similar fashion when more than two paths are found between a given input and output.

With this in mind, the following outlines the delay expression generation process for each input of a subcircuit:

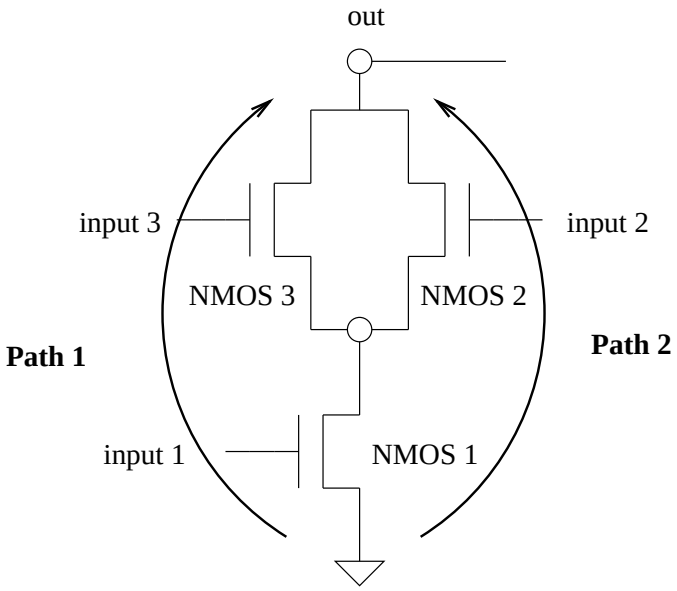


Fig. 5. Multiple Paths Example

1) For each transistor that is connected to an input, find all paths from that transistor to the output, and from that transistor to the ground or $V_{dd}$ terminals without passing through the output node. Here we assume that for each transistor it is always possible to find paths to output and paths to either one of ground or $V_{dd}$ terminals, but not both. We store the paths that we found in an array.
2) For each of the paths that are found, check if transmission gates exist in the path. This is done by examining each node in the path and check to see if a loop can be found through another transistor. If that is the case the path is marked with a special flag so that appropriate delay models can be used.
3) Generate the delay expression for each of the paths by considering the transistor stacks that exist in that path and the corresponding parasitic capacitance that is driven by the stack (see example below).
4) If multiple paths exist, combine each of the expressions. Determine the type of delay expression by checking whether the path terminates at ground or $V_{dd}$ terminal and the type of transistor that the input is connected to. Output the final expression to file.

As an example of delay expression generation consider the NMOS stack in Fig. 6. Input 1 is connected to a NMOS transistor stack with three transistors. We assume that both input 2

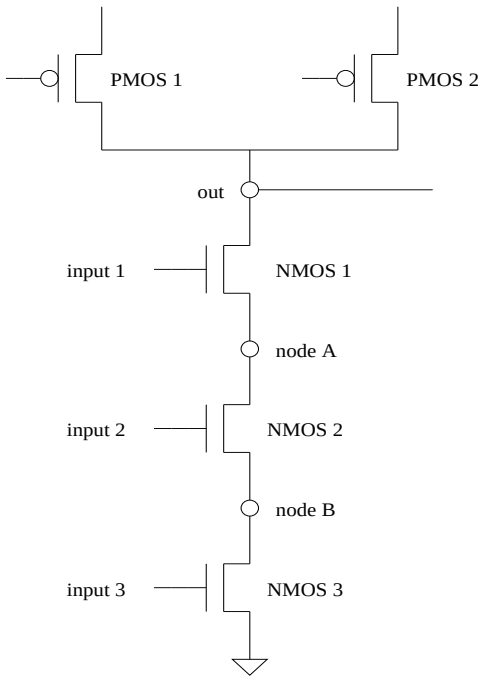


Fig. 6. Delay Expression Example

and input 3 were driven to $V_{dd}$ at an earlier time (otherwise the output would not be driven to ground) and have discharged the parasitic capacitance at nodes A and B respectively. Thus the only capacitance we need to consider is that of the output node, which consists of both parasitic and load capacitance as it is the output node of a subcircuit. The delay expression generated for input 1 will be:

$$(\text{3 stack NMOS model})(\text{parasitic cap. at output node + load cap. at output node}) \qquad (7)$$

Since load capacitance depends on subcircuit instance connectivity, it cannot be determined at this point when we are generating delay expressions for each subcircuit definition. Thus for load capacitance we insert the place holder '_LOAD_' so that actual capacitance information can be substituted at a later point in the solver.

However, for input 2, even if we assume that input 1 was driven to $V_{dd}$ earlier the parasitic capacitance at the output node still has not been discharged, as no electrical path from the output node to ground existed, thus in generating the delay expression for input 2 we need to consider the parasitic capacitance of the output node, which is discharged by a three stack NMOS transistor after input 2 is driven to $V_{dd}$, and the parasitic capacitance of node A that

is discharged by a two stack NMOS transistor. We continue to assume that node B has been previously discharged by NMOS 3, and the final parasitic capacitance expression is then:

$$(\text{3 stack NMOS model})(\text{total cap. at output node}) + (\text{2 stack NMOS model})(\text{parasitic cap. at node A}) \quad (8)$$

The same argument can be applied for input 3 to obtain the following delay expression:

$$(\text{3 stack NMOS model})(\text{total cap. at output node}) + (\text{2 stack NMOS model})(\text{parasitic cap. at node A}) + (\text{1 stack NMOS model})(\text{parasitic cap. at node B}) \quad (9)$$

Given the delay expressions, the solver can formulate the delay equations at each subcircuit instance node as discussed in Section III for area optimization. However, in order to perform power optimization we still need the activity factor of each node as discussed in the next section.

## IV. Activity Factors Generation

While optimizing under a timing or a chip area constraint requires only the delay expressions of each subcircuit, for power optimizations switching information of each circuit node is also needed. In CMOS circuits, dynamic energy dissipation can be estimated as the following:

$$E_{\text{dynamic}} = \sum_{nodes} C_{node} V_{dd}^2 \, \alpha_{0\to1} \quad (10)$$

where $E_{\text{dynamic}}$ is the total energy dissipation of the design due to dynamic switching, $C_{node}$ is the capacitive load of each circuit node (both load and parasitic), and $\alpha_{0\to1}$ is the activity factor of each node from ground to $V_{dd}$.

### *A. Activity Factor Definition*

We define activity factor for each circuit node in the design, over a finite number of simulation runs, as:

$$\alpha_{estimated\ 0\to1} = \frac{\text{number of } 0 \to 1 \text{ transitions in } n \text{ operations}}{n} \quad (11)$$

This definition is an estimation of the transition probability of each circuit node, which is:

$$\lim_{n\to\infty} \alpha_{est.\ 0\to1} \quad (12)$$

While our definition is an estimation of the true probability, it nonetheless provides a practical method for obtaining the factors from simulation runs. In [8], Najm provides another approach where transition probabilities at each gate's output are calculated based on the transition probabilities of each input node. However, that approach requires knowledge about the logic function of each gate that needs to be generated before the probabilities can be calculated.

### *B. Simulation Methodology with IRSIM*

Activity factors are generated in our framework as follows:

1) As discussed in Section II-D, in the circuit analyzer, the primary inputs of the design are extracted from the information provided by the parser. These nodes represent the input interface that the design provides to the external circuitry and are retrieved from the internal representation generated by the parser.
2) We combine all primary inputs together as an IRSIM vector, and we generate as many stimuli on the input vector as specified by the user. Each stimulus is randomly generated from random number generator and is recorded in the IRSIM input command file.
3) In addition to generating input stimuli, we also note in the IRSIM input command file that we would like IRSIM to report the status of each subcircuit instance node after each operation. The IRSIM command file, along with the circuit design (after being translated from SPICE netlist format into IRSIM format by a separate tool), is passed into IRSIM for simulation.
4) We record down the output from IRSIM into a file. The IRSIM output is of the form:

```
time = <time1>
node1 = 1 node2 = 0
...
time = <time2>
node1 = 0 node2 = 0
...
```

The output file is parsed and we record the status of each node at the end of each operation. After the parsing is finished, we iterate through each node, count the number of transitions, and calculate the activity factors from Equation 11. The results are then printed to an output file as {node name, activity factor} pairs.

### C. 32-bit Adder Example

As an example of activity factor generation, we used the Ladner-Fischer adder design [6]. In this simulation, we counted both transitions from ground to $V_{dd}$ and $V_{dd}$ to ground (which is twice of the activity factor that we defined in Eq. 11).

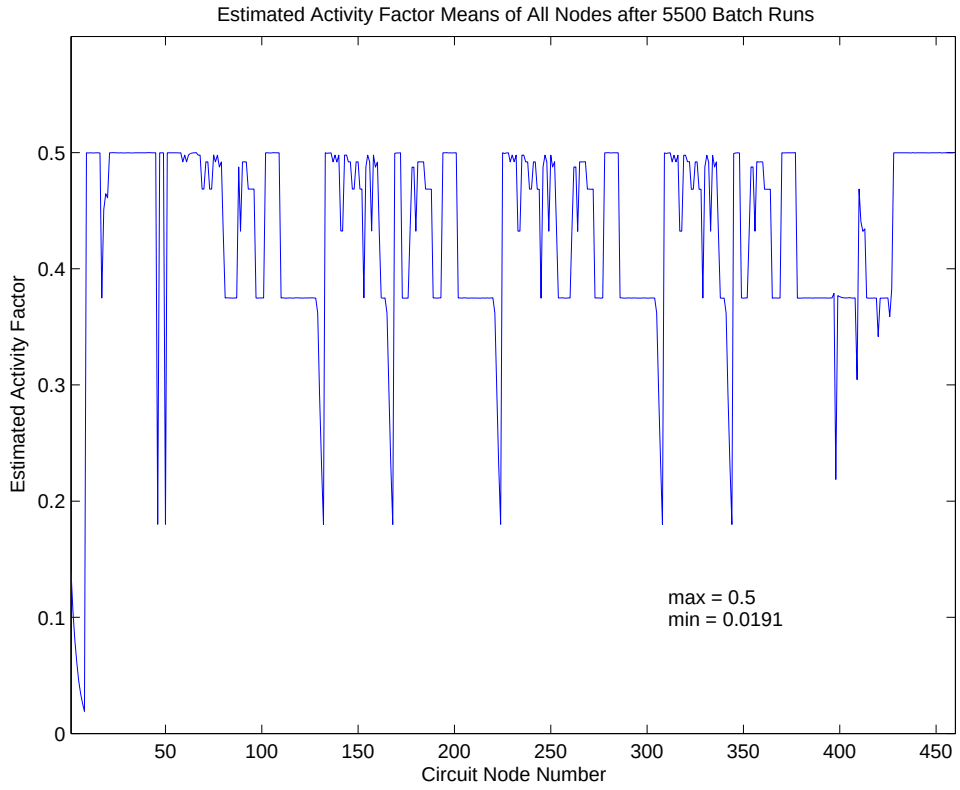


Fig. 7. Means of Activity Factors over 5500 Batch Runs

Figure 7 was generated after running 5500 batch simulations, with each simulation being 10000 operations (with a time step defined to be 1000 ns). As a measure of converging rate of our estimation to the real transition probability, we ran another batch of simulations with the number of time steps equals to the batch number. The results show that our estimated activity factor converges very closely to the true transition probability after about 2000 operations.

## V. Conclusion

This thesis presents a circuit optimization framework based on a circuit parser and analyzer together with a mathematical optimizer. We described the implementation of a prototype of our

framework and presented some preliminary results.

A number of future directions can be taken in our work. On circuit modeling, more accurate models can be generated for each transistor stack and we can also expand the models to include different circuits such as domino ones. On circuit parsing and analysis, the parser can be expanded to be able to recognize more different types of circuit structures besides the transmission gate. The limitation of each subcircuit being a channel connected component can also be eliminated with a netlist flattener, which flattens the input netlist into transistors instead of interconnected subcircuits, followed by a circuit recognizer that constructs channel-connected components from the flattened netlist. Special structures such as the transmission gate can then be identified on each of the recognized components and the delay expressions be generated appropriately. In power estimation, we can include an implementation of the probability propagation scheme from [8] and compare the time required to generate the activity factors with our estimation scheme.

## REFERENCES


[1] EinsTuner. `http://www.eetimes.com/story/OEG20020604S0056`.

[2] Elmore, W.C., "The Transient response of Damped Linear Networks with Particular Regard to Wireband Amplifiers." In *Journal of Applied Physics*, Vol. 19, pp. 55-63, January 1948.

[3] Fishburn, J.P. and Dunlop, A.E., "TILOS: A posynomial programming approach to transistor sizing." In *IEEE Transactions on Computer-Aided Design of ICs and Systems*, Vol. 11, pp. 1621-1634, November 1993.

[4] GNU flex. `http://www.gnu.org/software/flex`, and GNU bison `http://www.gnu.org/software/bison/`.

[5] IRSIM. `http://bwrc.eecs.berkeley.edu/Classes/IcBook/IRSIM/`.

[6] Knowles, S., "A family of adders." In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pp. 277-284, June 2001.

[7] Mosek. `http://www.mosek.com`.

[8] Najm, F.N., "Transistion Density: A New Measure of Activity in Digital Circuits." In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 2, pp. 310-323, February 1993.

[9] Pathmill. `http://www.synopsys.com/products/analysis/pathmill_ds.html`.

[10] Pelgrom, M.J.M., Duinmaijer, A., and Welbers, A., "Matching properties of MOS transistors." In *IEEE Journal of Solid-State Circuits*, Vol. 24, pp.1433-9, October 1989.

[11] Toh, K., Ko, P., and Meyer, R.G., "An engineering model for short-channel MOS devices." In *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 4, pp. 950-958, August 1988.