# TLG 0.7 Beta:
# The Transistor-Level Generator

**A Design Project Report**

**Presented to the Engineering Division of the Graduate School**

**of Cornell University**

**in Partial Fulfillment of the Requirements for the Degree of**

**Master of Engineering (Electrical)**

**by**

**Paul J. Grzymkowski**

**Project Advisor: Rajit Manohar**

**Degree Date: May 2003**

**Abstract**

Master of Electrical Engineering Program

Cornell University

Design Project Report

**Project Title: TLG: The Transistor-Level Generator**

**Author: Paul J. Grzymkowski**

**Abstract:** The purpose of this Master of Engineering project is to design and develop tools from the ground up in C to automatically generate transistor-level layout in MAGIC from production rules in CAST. The main focus is the automated design of complete bit-level layout through stack generation, source/drain/gate wiring, and multilayer wiring algorithms. The ultimate goal is the ability to break down the functional units of a microprocessor's production rules, run them on multiple instances of TLG concurrently, and arrive at a nearly complete transistor-level layout of the microprocessor.

To generate transistor-level layout TLG receives desired nfet and pfet structures, as well as definitions from input files. TLG analyzes the structures and connects as many of the transistors as it can with direct source and drain connections, generating transistor stacks. TLG shifts the position and height of the stacks to maintain functionality and use area more effectively. Additional considerations are made for connections that are internal nodes, output nodes, or power rails. From there TLG analyzes the stack structure and attempts to wire the remaining source and drain connections, as well as all of the gates. A shortest-path algorithm is implemented to generate wires of minimal length while observing all appropriate design rules. Wiring begins on the same metal layer as the desired connection, but may move to alternate metal layers if the path is deemed more suitable. Additional considerations are made to add contacts between metal layers, specify general metal layer orientations, or avoid metal layers entirely. The end result is output directly to a MAGIC file and is able to be immediately read and modified as desired.

**Report Approved by**

**Project Advisor: _____ Date: _____**

## Executive Summary

The purpose of this Master of Engineering project is to design and develop tools from the ground up in C to automatically generate transistor-level layout in MAGIC from production rules in CAST. The main focus is the automated design of complete bit-level layout through stack generation and wire routing algorithms. The result is TLG: The transistor-level generator.

To generate transistor-level layout TLG receives desired nfet and pfet structures, as well as definitions from an input TLG file. TLG analyzes the structures and connects as many of the transistors as it can with direct source and drain connections, generating transistor stacks. Additional considerations are made for connections that are internal nodes, output nodes, or power rails. From there TLG analyzes the stack structure and attempts to wire the remaining source, drain, and gate connections. The Maze Routing algorithm is implemented to generate wires of minimal length while observing all appropriate design rules. Wiring begins on the same metal layer as the desired connection, but moves to alternate metal layers if the path is deemed more suitable. The end result is output directly to a MAGIC file and is able to be immediately read and modified as desired.

Overall the project has been a success. Stacks are generated correctly, the output format is compliant with MAGIC, and most connections encountered can be wired. Substrate contacts, diffusion contacts, and labels are all intact as desired. TLG uses a number of special purpose data structures to compactly store and analyze the input transistor data, and it uses a modified version of the Maze Routing algorithm to effectively route wires. TLG is not without its limitations, however. Once the stack structure is generated it is permanent, which can prevent efficient wiring of certain connections later on. Additionally, a placed wire may be a minimal path to its destination, but it may prove to disrupt many subsequent wires. However, any shortcomings of TLG are far outweighed by the flexibility of the platform developed, the uniqueness of the algorithms used, and the ability for countless other features to be added easily in future versions of TLG.

# Table of Contents


**1. Introduction ..... 1**

**2. Design Specifications ..... 1**

**3. TLG Operation ..... 4**
- Overview ..... 4
- TLG File Format ..... 6
- MAG File Format ..... 8
- The `trans` Structure ..... 11

**4. Stack Generation ..... 13**
- Connection Graphs ..... 13
- Stack Walking ..... 14
- Stack Making ..... 17

**5. Wire Routing ..... 20**
- Wire Requests ..... 20
- Maze Routing ..... 22
- Routing Implementation ..... 23
- Route Checking ..... 26
- Multilayer Routing ..... 27

**6. Future Work ..... 29**
- Improvements ..... 30
- New Features ..... 30

**7. Results and Conclusions ..... 31**
- TLG Operation ..... 31
- Stack Generation ..... 32
- Wire Routing ..... 32
- Final Thoughts ..... 33

**8. Acknowledgments ..... 35**

**9. References ..... 35**

**10. Appendices ..... 36**
- TLG Code Summary ..... 36
- TLG File Format Definitions ..... 39
- MAG File Available Layers ..... 39
- Sample .wire and .grid Outputs ..... 40
- Sample TLG Outputs ..... 41
- TLG Operations Guide ..... 48

# 1. Introduction

Transistor-level layout is an essential step in the design of any microprocessor. While truly an art form at its zenith, manual transistor layout is usually considered to be a monotonous, tedious, and ultimately incredibly time-consuming exercise. Like so many innovations of the past, transistor-level layout is a perfect candidate for computer automation. Enter TLG: the Transistor-Level Generator.

The prevailing goal of TLG is to alleviate, if not completely eliminate the pain of manual transistor layout. Nestled softly in between the production rule stage and datapath design stage of microprocessor development, TLG receives input files representing the cell level production rules of the design and any other specific definitions. TLG generates intermediate representations of the desired transistor structures to establish interconnections, generate transistor stacks, and provide shortest-path wiring between source, drain, and gate connections. Ultimately, TLG generates an output file that contains the complete cell layout according to the specified production rules and design rules.

It is hoped that TLG will save students and other chip designers time that can be better allocated elsewhere to other endeavors, such as making their processors actually work. TLG reduces the necessity to spend long hours drawing and redrawing transistors, wires, contacts, and the like, and instead merely requires a few input parameters specified at generation time, or in the input files themselves. The ultimate purpose for TLG lies in the possibility of breaking down the functional units of a microprocessor's production rules, running them on multiple instances of TLG concurrently, and arriving at a nearly complete transistor-level layout of the microprocessor.

# 2. Design Specifications

TLG was developed partially under the premise that it would ultimately be used in part by students at Cornell University required to layout their VLSI projects for courses such as ECE 474: Digital VLSI Design and ECE 574: Advanced Digital VLSI Design. As

such, TLG was intended to be compatible with the existing tools those types of courses utilize, namely CAST: a production rule editor [1], and MAGIC: a VLSI layout editor [2]. Thus, TLG was to be developed from scratch in C to automatically generate MAGIC compatible output from CAST compatible input. TLG's place in a student's microprocessor design cycle was then determined to potentially be between the production rule stage and datapath design stage as shown in **Figure 1** below.

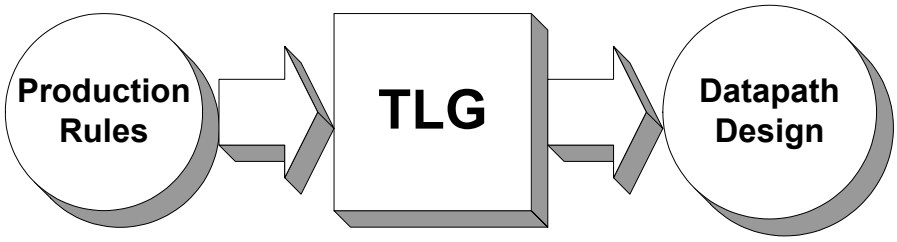



**Figure 1:** TLG Position in a Typical Design Cycle

Existing tools within MAGIC provided the preliminary ability to generate nfet and pfet stacks of transistors of specified sizes from production rules through direct source and drain connections. Thus, developing TLG to this level was both an obvious initial goal, and an evolutionary point that clearly needed to be exceeded. It was first established that additional considerations should be made for stack connections that are internal nodes, output nodes, or power rails. The next main objective was to analyze the stack structure and wire the remaining source, drain, and gate regions of the resulting stacks together so they could be directly simulated in IRSIM or ASPICE and indicate more precisely the ultimate behavior of the circuit. It was clear that these generated wires needed to be relatively optimal, and observe all the necessary design rules, so a careful selection of routing algorithms would be needed. From this point, it was clear that the wiring procedure should also be extended over multiple metal layers to ensure more flexibility and hopefully more optimal wiring. It was also determined that key parameters should be provided into all of these processes to decide which metal layers should be used, in what orientations, and with what precedence.

The project had obvious extensions in nearly every direction, including arraying single cell layout blocks, wiring existing MAGIC layout files, receiving input directly from technology files, and more advanced optimization techniques. It was determined that these features, while both useful and relevant, would not make it into the current version of TLG. However, many of these additional features have been anticipated for future versions, and aspects of TLG have been designed to facilitate a smooth implementation of these features. A more detailed discussion of some of these issues can be found in the **Related Work** section near the conclusion of this report.

With a firm grasp of the project time constraints, the development of the current version of TLG was focused exclusively on single cell level layout. From there a set of specifications and project milestones was created. The following bullets in **Table 1** summarize the resulting set of goals established:

**Table 1:** TLG Goals and Specifications

**Basic Operation**

- Read `tlg` files as input for production rules
- Output files in `mag` (MAGIC) format
- Generate transistors with labels
- Parse specified lengths and widths, and partial inputs
- Parse key specifications as definitions from input files

**Stack Generation**

- Merge series and parallel transistors together
- Generate transistor connection graph
- Insert contacts into required sections
- Recognize and generate loop stacks

**Wire Routing**

- Break stack structure apart into grid
- Connect source to destination using a search queue
- Retrace minimal path wire and add to `.mag` structure
- Route using wire spacing considerations
- Source, drain, and gate connections
- Use additional metals for multilayer routing
- Provide vias and contacts between layers
- Constrain metal layers and allow precedence
- Maintain design rule specifications

# 3. TLG Operation

**Overview**

In its most primitive form TLG takes in an input file, performs operations on it, and produces an output file. The input `tlg` file represents the production rules from CAST and parameters to modify the analyses. The operations include parsing the input file, generating intermediate representations to perform transistor stacking, wiring routines and optimizations, outputting intermediate `grid` and `wire` files, and formatting the results to be output while printing them to the screen if desired. The resulting `mag` file represents the corresponding transistor layout and is directly readable with MAGIC. A basic block diagram of the data flow of TLG is seen in **Figure 2** below. A summary of all the global variables and procedures used in TLG can be found in **Appendix A**.

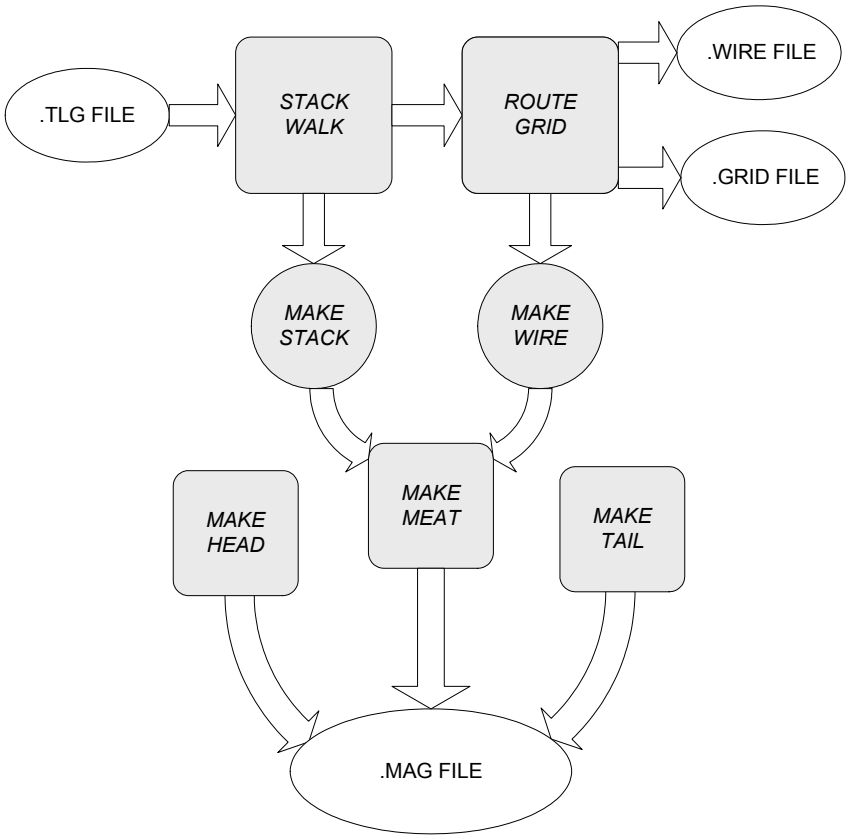



**Figure 2:** TLG Basic Data Flow

In the beginning of TLG's procedure, after all data structures are initialized, command line parameters are checked to setup global variables. The procedure `makehead()` is called to write the beginning of the output `mag` file that will be built throughout the execution of TLG. For a discussion of this format see the **MAG File Format** section that follows. Next `getmeat()` is called which parses the input `tlg` file identifying nfet and pfet transistors and their optional characteristics, including oxide length and width, and source, drain, and gate labels. More about this file format is explored in the **TLG File Format** section. According to these identifications `getmeat()` creates `trans` structures: general-purpose structures that represent all necessary aspects of a transistor. These structures are explored in **The `trans` Structure** section that follows this discussion. In the process of creating these structures `getmeat()` adds them to the statically defined `fets[]` array. Immediately after this `getmeat()` calls `locate()` to determine if the newly created transistor should be connected to an existing one, namely if its source,

drain, or gate label matches a previously defined label. If there is a match, and the transistors are of the same type (n-type or p-type), a pointer in the previous transistor is set to the new transistor, indicating a parent-child relationship between them. This process is continued over all transistors.

At this point in the execution of TLG the stack procedure is run. This stage will be discussed in greater detail in the **Stack Generation** section, but generally it involves walking through the `fets[]` array to find possible stack heads, then walking through the children of the head to build the rest of the stack. When this process is complete, `makestack()` is run which generates data for layout array structures based on the generated stacks, creates diffusion contacts for needed areas, well plugs if desired, and flags and queues connection points that need wires.

The next step for TLG is to run the wiring procedures. These too will be discussed in great detail in the **Wire Routing** section but in general involve filling a 2D or 3D grid with the current stack information via the procedure `printgrid()`, and fulfilling all the wiring requests one by one using `routegrid()`. After each new wire is created a procedure `makewire()` is called to add its layout data to the existing set. The final phase of TLG's execution is calling the procedure `makemeat()` which dumps all of the layout data collected to the `mag` file, along with `maketail()` which adds the epilogue and closes the file.

### TLG File Format

The initial concept for TLG involved reading the desired production rules directly from a CAST file. It was decided, however, that the main focus of the project was on the automated layout and optimizations, and creating a good production rule parser was a formidable undertaking within itself. Removing this requirement allowed for greatly increased flexibility in the input format of the netlist, and so `tlg`, a new and straightforward file format was created.

The `tlg` file format is line based, insofar as each line indicates one completely variable definition. If the line begins with the identifier "`nfet`", "`pfet`", or "`defs`" the entire line is interpreted as an nfet transistor, pfet transistor, or definition respectively. Additionally, the existence of a pipe symbol "|" anywhere in the line taints it, effectively, and causes the entire line to become a comment. TLG currently interprets only the first 100 characters of a line for no good reason other than to make the length fixed, and this value can be increased or decreased as desired.

If an nfet or pfet are specified in a `tlg` file the remaining line is devoted to the transistors characteristics, all of which are optional and do not need to be specified in any particular order. The characteristics include oxide length and width, and source, drain, and gate labels. If transistor length and width are omitted default values are assumed. Labels are assumed to be inherently unique, and non-unique labels are assumed to be intentionally connected. Thus, unique identifiers are created for the unlabeled connections, and they are assumed to be disconnected from all other nodes. Labels ending in "!" are assumed to be power rails, while labels ending in "#" are assumed to be internal nodes (nodes that require no label or connection, but exist as internal representations of connections). Nodes of these two special cases are handled differently by the stack generation procedure. Transistor characteristics are signified by a single letter followed by a colon. The identifiers "l:", "w:", "s:", "d:', and "g:", represent the transistor's length, width, source, drain, and gate, respectively. To define a simple CMOS inverter, as in **Figure 3a**, for example, the `tlg` file shown in **Figure 3b** could be written.

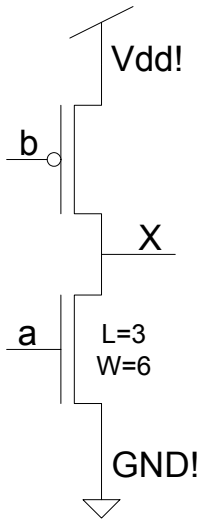



**Figure 3a**: A CMOS Inverter

```
| inv.tlg - a simple inverter |
nfet l:3 w:6 s:GND! g:a d:X
pfet d:X g:a s:Vdd!
```

**Figure 3b:** Sample `tlg` File for an Inverter

Note that in this example the length and width of the nfet are specified to be 3 lambda and 6 lambda respectively. The length and width of the pfet are unspecified and will be set to the current default values of 2 and 5 respectively.

If "`defs`" is specified in the beginning of a line in a `tlg` file the remainder of the line is used to setup initial declarations and design rules. These definitions include n-difffusion contact width, minimum n-diffusion to p-diffusion width, metal 1 wire width, etc. If an incomplete set of definitions is specified, or no definitions are specified the default values will be assumed. A listing of currently available definitions, as well as their default values can be found in **Appendix B**.

Clearly other nuances can be added to the `tlg` format to better define the information being transferred from the production rule phase, they are just not handled by the current version of TLG.

**MAG File Format**

The magic format is relatively straightforward to generate, and as such a complete description is not included here. Rather, this section focuses on the techniques used by

TLG to output compliant files. For full documentation of the MAG format please see "Magic – Format of `.mag` Files Read/Written by Magic" [3] in the **References** section.

In general a `mag` file has three main sections: the header, body, and footer. The header is comprised of the identifier "`magic`", signifying it is a Magic file, an optional specifier of the technology being used, and an optional timestamp indicating when the file was last edited. For single cell layout, the body of the file contains mask rectangles divided by layer, and labels. Finally, the footer contains a line signifying the end of the file. A sample magic file representing a single nfet transistor with labels is shown in **Figure 4** below.

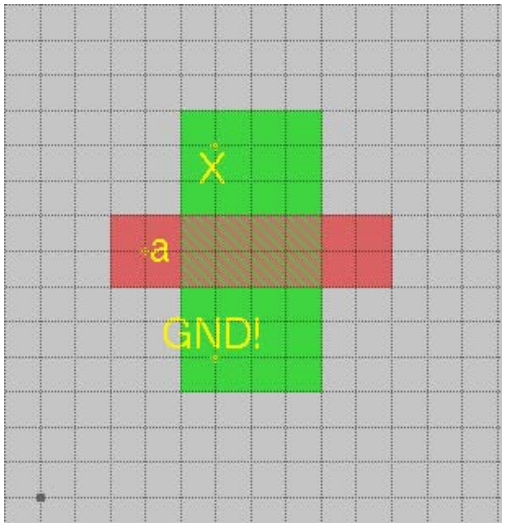



**Figure 4a**: Transistor Layout in Magic

```
magic
tech scmos
timestamp 1052873976

<< ntransistor >>
rect 4 6 8 8
<< ndiffusion >>
rect 4 8 8 11
rect 4 3 8 6
<< polysilicon >>
rect 2 6 4 8
rect 8 6 10 8
<< labels >>
rlabel polysilicon 3 7 3 7 3 a
rlabel ndiffusion 5 4 5 4 1 GND!
rlabel ndiffusion 5 10 5 10 5 X

<< end >>
```

**Figure 4b:** Sample `mag` File for a Transistor

Each layer of the body of the `mag` file is signified by "<<" and ">>" symbols. These layers directly correspond to their physical equivalents: the n-type transistor oxide ("ntransistor"), n-type diffusion ("ndiffusion"), etc. A list of the available layers in the `mag` file is included in **Appendix C**. The `mag` structure uses rectangles to represent patches of particular material layers, and indicates the absolute position of the rectangles with lambda coordinates. The rectangle is specified by "`rect`" and includes four fields to define the lower left corner (xbot, ybot) and upper right corner (xtop, ytop) of its position in lambdas from the origin. The format of a rectangle is:

**rect** *xbot ybot xtop ytop* [3]

Similarly labels are defined with rectangles but have the added information of the layer the label corresponds to, the position of the text with respect to the label, and the text of the label. The label is specified by the identifier "rlabel", and the format is:

**rlabel** *layer xbot ybot xtop ytop position text* [3]

TLG calls the procedures makehead() and maketail() to generate the necessary information for the header and footer of the `mag` file. To represent rectangles a `rect` structure was created with four position fields. To represent labels a `label` structure was created with the fields described above. These structures are summarized in **Table 2a** and **Table 2b** below.

**Table 2a:** The `rect` Structure

| Field | Type | Description |
|---|---|---|
| **xbot** | `int` | X coordinate, bottom |
| **ybot** | `int` | Y coordinate, bottom |
| **xtop** | `int` | X coordinate, top |
| **ytop** | `int` | Y coordinate, top |

**Table 2b:** The `label` Structure

| Field | Type | Description |
|---|---|---|
| **layer** | `char *` | Layer name |
| **r** | `rect` | Position rectangle |
| **pos** | `int` | Text position |
| **name** | `char *` | Label text |

The `rect` structure is used throughout TLG to store layout information in arrays. In general, a static array is allocated for each type of layer, along with a counter indicating how many elements have been placed in the array. When a `rect` is added to an array during stack generation or wire routing, the counter is incremented and the parameters of the `rect` are defined. After all generation routines are complete the procedure `makemeat()` is called which goes through each layer array and writes the layer header

and `rect` information to the `mag` file. Labels are handled similarly, but are only generated during the stack generation phase.

### The `trans` Structure

As mentioned before, the `trans` structure represents all necessary aspects of a single transistor. The `trans` structure follows directly from the line inputs from the `tlg` file, as one would imagine since they are so closely correlated. Additionally, the structure has been augmented with pointers and counters to represent the hierarchical information dictated by the interconnections between transistors. The `trans` structure fields are summarized in **Table 3** below. Note that each of the three types of connections (source, drain, and gate) have their own set of pointers and counters.

**Table 3:** The `trans` Structure

| Field | Type | Description |
|---|---|---|
| **l** | `int` | Oxide length |
| **w** | `int` | Oxide width |
| **s** | `char[]` | Source name |
| **sleft** | `trans *` | Source pointer, left |
| **sright** | `trans *` | Source pointer, right |
| **sout** | `Int` | Source connect counter |
| **d** | `char[]` | Drain name |
| **dleft** | `trans *` | Drain pointer, left |
| **dright** | `trans *` | Drain pointer, right |
| **dout** | `int` | Drain connect counter |
| **g** | `char[]` | Gate name |
| **gleft** | `trans *` | Gate pointer, left |
| **gright** | `trans *` | Gate pointer, right |
| **gout** | `int` | Gate connect counter |
| **color** | `int` | Color (used vs. unused) |
| **type** | `char` | 0: n-type, 1: p-type |

A static number of `trans` are allocated in the `fets[]` array with a counter representing how many transistors are being used. When the input `tlg` file is read using `getmeat()` the counter is incremented and new `trans` structures are created. At this point the l, w, s, d, g, and type parameters are known or set to be the default values. The parameter color, used mainly for stack generation, is set to zero. The procedure `locate()` is then

called to compare the source, drain, and gate names to any previously defined transistors in order to build a connection graph (CG) between them. If any transistor with a source, gate, or drain of the same name exists, the two transistors will ultimately need to be connected together, and the counter (sout, dout, or gout) of both transistors are updated to reflect the connection. If the dout of a transistor is 3, for example, it indicates that there are two additional connection points that share the same name as the transistor's drain and will need to be connected in the layout. Connections between transistors of different types will need to occur through wires (since the n-diffusion and p-diffusion regions are not allowed to be close to each other), but connections between transistors of the same type may be able to occur directly by stacking them together. Thus, if `locate()` detects a match between transistors of the same type, the appropriate xright pointer in the found transistor is set to point to the new transistor, while the appropriate xleft pointer in the new transistor is set to point back to the older one. This pointer assignment creates a doubly linked list of transistors according to connection name, with left pointers always pointing left from newer transistors to older ones, and right pointers always pointing right from older pointers to newer ones. A diagram showing a possible pointer configuration for a `tlg` file with three nfet transistors is shown below in **Figure 5**.

```
| threetrans.tlg – 3 fets |
nfet s:GND! d:Y# g:c
nfet s:Y# d:X g:b
nfet s:Y# d:X g:a
```

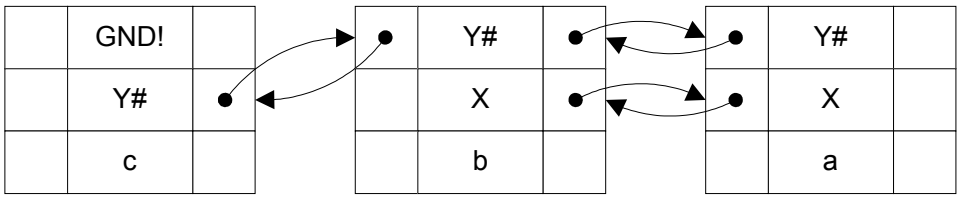



**Figure 5:** Pointer Connections for Three Nfets

While having a doubly linked list makes the transistor structure more complicated, having pointers in both directions proves to be essential because it allows either transistor

to be immediately reachable from the other one. This quality is especially useful during stack generation.

## 4. Stack Generation

While wires are required to connect between transistors of different diffusion types, transistors that are both n-type or p-type and share a connection point can be directly connected through stacking. When wiring the three transistor example initiated in **Figure 5**, instead of just wiring them directly as shown in **Figure 6a** below, the sharing of the Y# and X connections can be exploited to stack the transistors on top of each other as shown in **Figure 6b**.

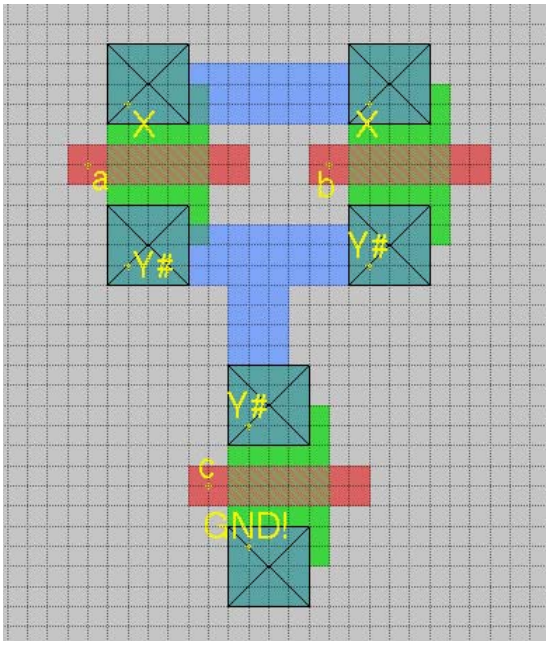



**Figure 6a:** Wiring of Three Nfets

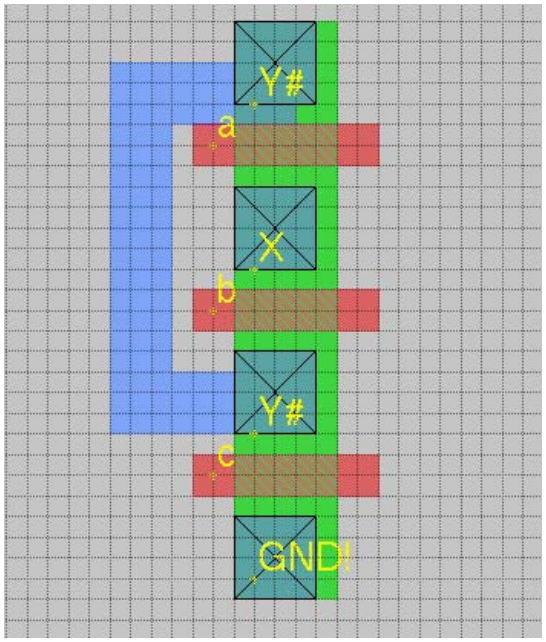



**Figure 6b**: Stacking of Three Nfets

The splitting apart of Y# requires an additional wire, but the resulting circuit is more compact, and is much less complicated to generate automatically. The motivation of the stack generation phase of TLG is to detect these scenarios and exploit them, if possible.

### Connection Graphs

The stack generation phase of TLG takes the `fets[]` array of interlinked `trans` structures and interprets them as connection graphs to determine how to generate stacks. Conceptually connection graphs depict a transistor as an edge, and a connection between a source and source, drain and drain, source and drain, or drain and source as a named point. Since gate connections are not considered during stacking, they only appear as names on the edges of the transistors. More advanced stacking techniques might consider gate connections to shift the position of stacks around to better facilitate gate wiring, but this feature has not been focused on as of yet. For a given circuit there may be more than one connection graph, if the circuit has multiple sets of completely independent transistors. However, in most cases only one connection graph will exist because the power rails connect all the transistors. An example connection graph for the three transistor example above is shown in **Figure 7** below. More complicated graphs can be generated depending on the input, but they still conform to the guidelines described earlier.

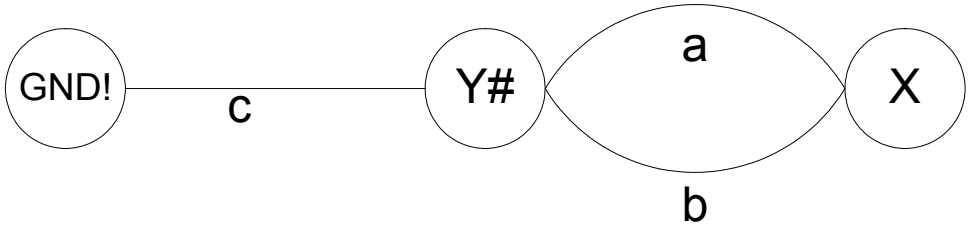



**Figure 7:** Three Nfet Connection Graph

The left and right pointers of the TLG `trans` structure are used to determine how many edges should connect to each point, while the connect counters keep track of how many edges are connected. These details become important during the stack walking phase, as described in the following discussion.

### Stack Walking

Once the connection graph for a given transistor topology is generated a relatively systematic approach dictates how to begin generating stacks. A stack can be generated from a connection graph by picking a starting node and walking from point to point along

unique edges until there are no more edges to move. A point can be visited multiple times, but edges can only be visited once. If there are remaining points in the connection graph that have not been reached yet, the process is repeated.

A stack will logically need to have a beginning and an end, namely a head transistor and a tail transistor. To determine an appropriate starting point the TLG procedure `printstack()` is called which searches through the `fets[]` array to find a potential head. If a node in the connection graph has an odd number of connections (namely if its sout or dout is odd) it means it will have to be a starting point or end point of a stack. When walking through the control graph the node will essentially become a "dead end" after the other edges are removed. These types of nodes become good candidates for being selected as the head of a stack.

In general it is more desirable for a power rail (i.e. GND! or Vdd!) to be made the head or tail of a stack because they are shared over the entire chip and thus have more external connections. If a power rail is not used another type node will need to be split to make the head. In this instance it is more desirable for internal nodes to become the head (instead of output nodes) to reduce the output capacitance that results from adding wires to connect the split node back together. Maintaining these preferences for head selection can be enabled or disabled in TLG as desired.

Once a head node is selected using `printstack()` the procedure `stackwalk()` is used to build the rest of the stack. The procedure follows the pointers of that particular trans, effectively following the edges of the connection graph. As a transistor is visited, its color is changed from 0 to 1 indicating it has been removed from the graph and should no longer be considered. When a transistor is visited, it is also spun into a new structure: `strans`, a stack transistor. The `strans` structure is summarized in **Table 4** below.

**Table 4:** The `strans` Structure

| Field | Type | Description |
|---|---|---|
| **t** | `trans *` | The transistor itself |
| **color** | `int` | Transistor orientation |
| **left** | `strans *` | Stack pointer, left |
| **right** | `strans *` | Stack pointer, right |

New `strans` structures are stored in `stacks[]`, a statically allocated array. The `stacks[]` array increases the storage requirement of TLG, but it is beneficial because it is a simpler representation of the stack structure than adding more `trans` fields. The s`trans` structure contains the transistor itself, as well as a pointer in each direction. The pointers maintain the stack as a doubly linked list. The head s`trans` thus has a left pointer set to NULL and the tail has a right pointer set to NULL. Lastly, a color field has been added to additionally define the orientation of the transistor in the stack, namely 0 if the source side is pointing down or 1 if the drain side is pointing down. This information is important because switching the orientation effects the way the source and drain connections are made. If `stackwalk()` reaches a transistor it cannot exit from it is made a tail node of the stack, and the process is repeated on a new head until there are no transistors remaining in the graph. In addition to generating s`trans` structures, `printstack()` optionally prints the stack to the screen in a suitable representation.

One of the remaining subtleties of this process is the detection of loop stacks; namely transistors that are connected in a ring and thus do not have any nodes with an odd number of connections. TLG detects these cases after detecting all the previous ones by picking a head from the remaining nodes (regardless of sout or dout) according to the node preference scheme described earlier. One additional subtlety is the process of removing a node from the connection graph. Since nodes of the same name are connected by a doubly linked list structure, removing a node would require retying together the pointers of the linked list. While this is certainly true, it does not tell the whole story because additional considerations need to be made over which *type* of connection is being made. Namely, if a source named Y of a transistor is connected to another transistor, it may be connected to the source, drain, or even gate of the other

transistor. As such, the name of each of the connections needs to be verified before the pointers are rewritten. This unfortunately hairy task is handled entirely by the procedure `transfix()`. Future versions of TLG may consider modifying the `trans` data structure to simplify this procedure.

### Stack Making

Once the entire connection graph has been painted and the `stacks[]` array has been filled the next step is to call the TLG procedure `makestack()`. This procedure walks through the stacks and generates layout for the appropriate `rect` arrays, including n-type diffusion, p-type diffusion, polysilicon, n-diffusion contacts, p-diffusion contacts, n-substrate contacts, p-substrate contacts, and metal layer 1 (`ndiff[]`, `pdiff[]`, `poly[]`, `ndc[]`, `pdc[]`, and `nsc[]`, `psc[]`, and `m1[]` respectively). The procedure starts the head at the lower left corner and generates the stack upwards. Every transistor gets its own oxide and polysilicon "ears" as well as a patch of diffusion. If a transistor is the head of a stack it gets an additional patch of diffusion. An annotated example of a stack generated from three transistors is shown below in **Figure 8**.

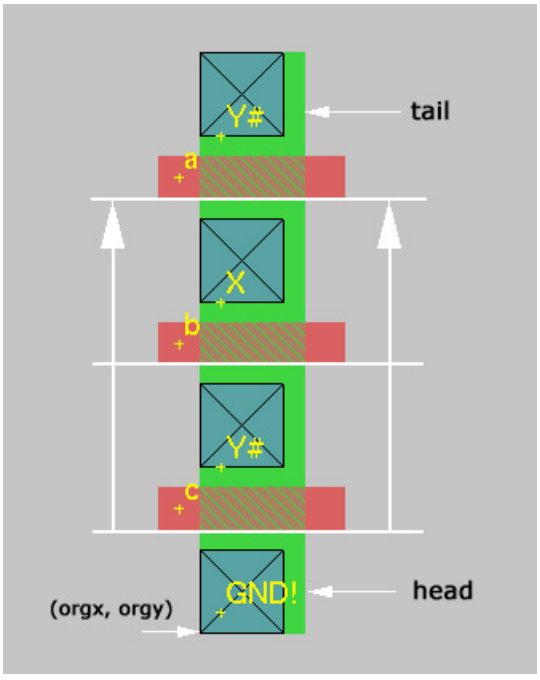
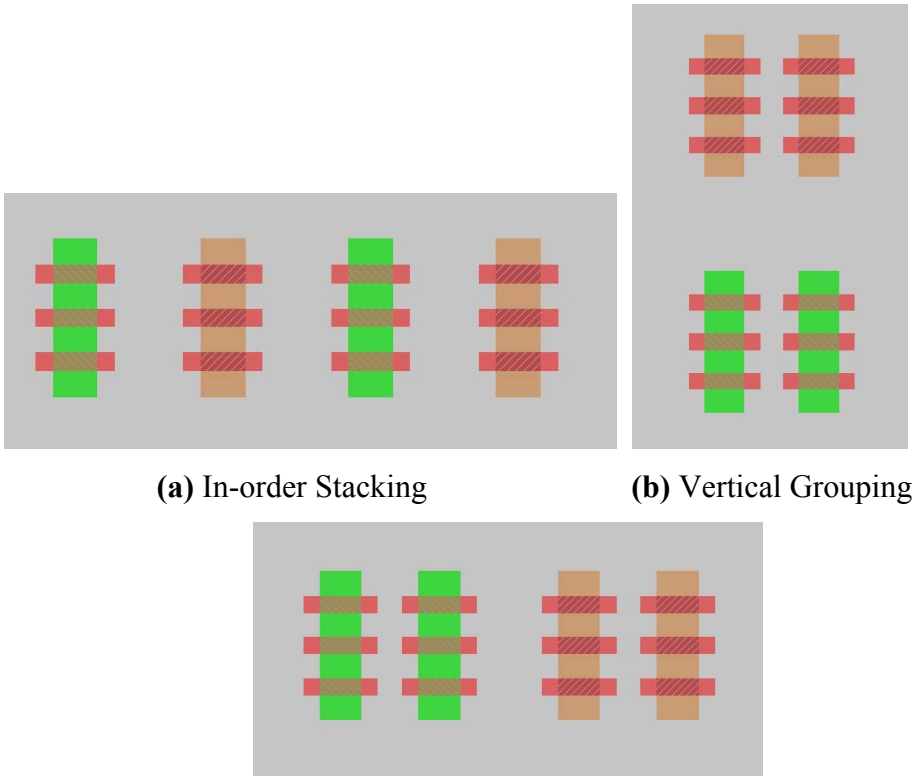



**Figure 8:** Three Nfet Stack Generation

At this point in the stack generation process the x and y coordinates of the rectangles are systematically determined. Global variables indicate where the next stack head should begin and from there positions are determined based on the user defined values and parameters particular to that technology. The values considered include transistor lengths and widths, minimum diffusion lengths, diffusion contact heights, and minimum poly overlap widths. When a stack tail is generated, the global pointers are offset to space the next stack from the current one. To adequately space the stacks apart, factors including poly-poly widths and diffusion-diffusion widths are considered. Additionally, if particular wiring schemes are selected more space may be added to better facilitate metal routing. Lastly, if a stack of a different diffusion type is to be placed, additional space is required between them to prevent design rule violations.

TLG is able to generate multiple types of stack positions. The default configuration places nfet and pfet stacks down in the order they were received in the `tlg` file, striping the ndiffusion and pdiffusion if necessary. This format is usually later beneficial for wire routing because `tlg` files tend to be written with forward connections; that is, an output of a particular section often connects to the input of a later section. In some instances, though, (and depending on how the `tlg` file is defined) it would be more beneficial to place the nfet stacks down first and then the pfet stacks creating two diffusion halves. Additionally, it may prove useful to place the pfet stacks above the nfet stacks. Any of these positioning options can yield acceptable results, and can thus be specified at TLG runtime, as described in **Appendix F**.



**(a)** In-order Stacking **(b)** Vertical Grouping

**(c)** Horizontal Grouping

**Figure 9:** Stack Positioning Options

Since this phase of the process is the last chance for transistors to be placed before wiring begins, it is an opportune time for `makestack()` to place necessary diffusion contacts, labels, and well plugs. A diffusion contact needs to be placed if the transistor is the head or tail of a stack, an output, or an internal node with more than two connections. These situations comprise nearly all cases, except those in which two transistors appear in series with an internal node in between them. In this case the diffusion regions can be shared. In general, labels are placed on all connection points. Labels for internal nodes can be optionally omitted anticipating that all unconnected nodes will be connected with a wire later. Unfortunately, this is not always the case, so it usually safer to keep the labels intact. Lastly, substrate contacts are optionally added to the regions that require them. The current methodology is somewhat primitive in that it only adds well plugs where there are power rails as the head or tail of stack. If a GND! connection exists as a head, a square of metal 1 is placed hanging off of the diffusion contact, and a substrate plug is placed off of the metal 1 square. This method is effective enough for relatively simple

transistor topologies but may prove to be inadequate for more complicated designs. A revised method should be sought for future versions of TLG.

In addition to generating the stack layout, diffusion contacts, labels, and well plugs as described above, `makestack()` also identifies connection points that require wires, a process which will be explained in the following section.

# 5. Wire Routing

With the transistor foundation firmly in place from the stack generation phase, the next step is to wire all the remaining connections together. While conceptually the objective is relatively straightforward, the implementation proves to be quite involved. In general, the process of wire routing involves first identifying what remaining connections need wires, then determining what these connections can be wired to, and finally trying to find an optimal path between them.

**Wire Requests**

Needed wires are identified during the TLG procedure `makestack`(). In general if a node has more than two connections (sout, dout, or gout = 3 or more) it will clearly need to be wired to another node, because only two nodes can be directly connected during stacking. To be able to detect needed wires for loop stacks this threshold is reduced to "more than one" connection (sout, dout, or gout = 2 or more). The exceptions to this rule are the internal nodes whose diffusion regions were merged. If the two connections were merged successfully, and there are no other connections to that node, there is no need for a wire. Lowering the threshold results in the occasional "false positive" of an output connection that does not actually need a wire, but for these instances the routing algorithm will realize there is nothing to connect the output to, and it will cancel the wire request. Once a wire is detected, a wire structure is generated and added to the wire request queue `wires[]`, a static array of wire structures. The wire structure is summarized in **Table 5** below.

**Table 5:** The `wire` Structure

| Field | Type | Description |
|---|---|---|
| **x** | `int` | Starting X coordinate |
| **y** | `int` | Starting Y coordinate |
| **s** | `int` | Stride |
| **name** | `char *` | Node name |
| **color** | `int` | Color (used, unused, etc.) |

A wire structure includes the x and y coordinates of the needed connection, a pointer to the name of the node, and the wire color to indicate if the connection has been made (it is initially set to zero). For simplicity, the point (x,y) corresponds to the lowest, leftmost corner of the desired connection. Thus for source and drain connections the lower left corner of the diffusion contact is signified, and for gate connections the lower left corner of the leftmost poly overlap is signified. Additionally, the wire structure has an interesting field referred to as the "stride" which corresponds to an offset in the x direction, and is intended to indicate an alternate starting point of the wire should the original x and y coordinates prove to be unwireable. In the case of a gate connection, the stride corresponds to the distance necessary to hop from the left polysilicon "ear" of the transistor to the right polysilicon ear. In the case of a source or drain connection the stride is the distance necessary to hop to the other side of a diffusion contact.

Once the stack structure has been thoroughly checked for needed wire connections, the next step is to figure out what they can connect to. The beginning stages of the TLG procedure routegrid() attempt to find a suitable destination for the starting connections. First an unused (color = 0) or partially used (color = 1) node is selected from the wire request array `wires[]`. It is then checked against all other wire requests, and if there is another connection with the same name that is completely unused `routegrid()` attempts to connect them together. If the connection is a success the original node is marked as used, and the node it was connected to is marked partially used. If the routing fails, however, `routegrid()` reexamines all the requests and attempts to connect the node to a partially used node of the same name. If this also fails, `routegrid()` uses the procedure `travwire()` to attempt to connect the node to any part of a wire electrically connected to a partially used node of the same name. If this final test fails, it either means there is no

good path to connect the node, or there are no other partially used nodes, which means the node is probably connected already.

At the conclusion of this analysis the needed wires have been identified, and their starting and ending points have been determined. The next step is to find an appropriate path to generate the wire, which requires a routing algorithm.

### Maze Routing

To connect a source node to a destination node TLG employs a modified version of the Maze Routing algorithm. This is a fairly common algorithm for routing wires in VLSI microprocessors and circuit boards. In this algorithm a metal layer is represented by a two dimensional grid. In the first phase of the algorithm the source node is expanded in all directions, with a counter signifying the current distance from the source node. As the "wavefront" from the source node is expanded over the grid, each move that is made is checked to ensure it is valid according to the design rules with respect to the transistor topology, existing wires, or other obstacles. The distance of each point that is successfully reached is recorded. If there are obstacles in the way, they may eventually be routed around, at the price of having a longer recorded distance. If the wavefront reaches the destination node, the path that represents the shortest distance is retraced backward to the source node. Finally, the extraneous information is removed from the grid, and the wire is generated. The stages of this process are depicted in **Figure 10** below.

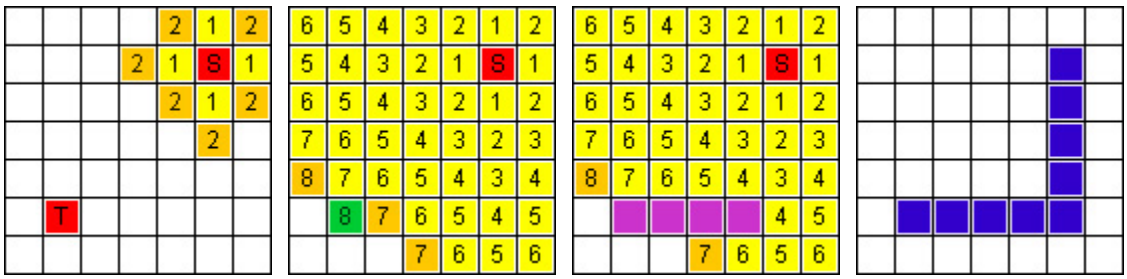



**Figure 10:** Maze Routing Algorithm
(adapted from Nestor, 2001) [4]

The Maze Routing algorithm was chosen largely for its simplicity and its ability to work for TLG's particular applications, but also because it guarantees an optimal path when routing a single wire. Note, however, that with the Maze Routing algorithm the optimal path of one wire may cause many suboptimal paths of subsequent wires, but this was considered out of the scope of this project. The focus of TLG was primarily generating correct, functional layout. Optimization has always been relevant, but a secondary priority. A possible way to combat the limitations of the Maze Routing algorithm would be to analyze the overall wiring efficiency over the entire circuit, and re-route wires that caused problems later on. This is a natural extension of the existing algorithm used by TLG, and could be considered for future versions.

**Routing Implementation**

TLG's implementation of the Maze Routing algorithm is a great deal more challenging than the ideal case described above, due in large part to varying sizes of metal widths and diffusion contacts, and various design rule considerations. To store the current status of the transistor layout, TLG allocates a two dimensional grid of integer values called `metalgrid[][]`, with one grid point representing one square lambda of layout. The grid is sized to encompass the entire layout with an adjustable pad around the perimeter to allow additional space for wiring. TLG also allocates an auxiliary grid called `wiregrid[][]` to store the progressive wiring data, which has a one-to-one correspondence with `metalgrid[][]`. A procedure called `printgrid()` is used to walk through all layout arrays and write to particular points in `metalgrid[][]` if that point should contain that particular material. Each material is assigned its own bit of the total value, so the existence of a particular material in a square lambda of layout (x,y) can be determined by simply performing an AND operation of metalgrid[x][y] with the desired material mask and observing the result. This encoding proved to be especially useful while checking for valid moves against design rules. At the conclusion of `printgrid()` a `.grid` file is created with a hexadecimal representation of the array. A sample intermediate output of this file is shown in **Appendix D**.

At this point in the routing process, TLG calls `routegrid()`. As described earlier `routegrid()` attempts to find a suitable connection point for a particular node by examining the unused and partially used nodes in the wire request array wires[]. From there it initiates the Maze Routing algorithm. The algorithm starts at point (x,y), the position of the specified source node and moves through `metalgrid[][]` in hopes of reaching the destination node. Each step of the way a procedure called `routecheck()` is executed to check if a move is valid. If so, `wiregrid[][]` is updated with the current distance count of the valid move using `routefill()`, and more moves into untouched areas are dispatched if possible. To reduce complexity the algorithm hops through `metalgrid[][]` with a step size equal to the current metal layer width. This is an optimization in some sense because it speeds up traversal through the grid, but at the same time it creates the added complication of, among other things, reaching destinations who are not perfectly aligned with the metal layer's stride. An alternate version of TLG uses single lambda strides when stepping through the grid, however the output is not currently reliable. This is clearly a feature that could be added to future versions of TLG.

Since the Maze Routing algorithm relies on breadth-first searching to generate optimal paths instead of depth-first recursion, one final structure was created: the `cell` structure, which is summarized **Table 6** below.

**Table 6:** The `cell` Structure

| Field | Type | Description |
|---|---|---|
| **x** | `int` | Current X coordinate |
| **y** | `int` | Current Y coordinate |
| **tx** | `int` | Target X coordinate |
| **ty** | `int` | Target Y coordinate |
| **m** | `int` | Current metal layer |
| **tm** | `int` | Target metal layer |
| **count** | `int` | Distance from source |
| **next** | `cell *` | Pointer to next cell |
| **dir** | `char` | Direction of motion |

The `cell` structure represents a set of work in the `wiregrid[][]` to be processed. It contains the current grid location (x,y) to be investigated, and the destination point (tx,

ty) that is intended to be reached. It stores the current distance count from the originating source (measured in metal hops), as well as the current direction of motion at the time of creation. The direction field was added to the cell structure to encourage routing to persist in a consistent direction, ensuring that wires are as straight as possible. The cell structure also stores the current and target metal layers, an addition used for multilayer routing which will be addressed later. Lastly, it stores a pointer to the next allocated cell. This field allows the cells to be queued up into a linked list structure when valid moves are discovered and dispatched in the order they are received, allowing a breadth-first execution of the Maze Routing algorithm.

A TLG procedure called addcell() is used every time a valid move in `metalgrid[][]` is found. It is used primarily as a constructor of cell structures, but also to link new cells to existing ones. A route queue is maintained through two global variables: `routequeue` which points to the head of the queue, and `routetail` which points, imaginably, to the tail. After a few initial cells are added with `routegrid()`, signifying the starting point for the algorithm, `emptyqueue()` is called to service the requests one by one. At this point, the procedure `routewire()` is called to act upon the current cell information. The procedure first checks to see if the route was found already, or if the current cell has come close enough to reaching the final destination node. If not, it calls `routecheck()` to examine the four spaces adjoining the current cell. If a move in a particular direction is deemed valid by `routecheck()`, the distance counter is incremented and a new cell is created with `addcell()`, to be serviced eventually by the `emptyqueue()` routine. In the fortunate event that the route was found, the coordinates where it was found and the distance count needed to get there are recorded. At any point in this process a `.wire` file can be generated from the current contents of `wiregrid[][]`. A sample intermediate output of this file is shown in **Appendix D**.

Once a route is found, the TLG procedure `retracewire()` is called to travel backwards over the optimal path. By using the final distance count, `retracewire()` looks for a path with successively descending values until the original source node is reached. The procedure looks on all sides of the current cell to see which cell should be next. As the

procedure traverses backwards through `wiregrid[][]`, it fills in the wire by overwriting the distance values with metal identification values. At the end of this process, the procedure `clearroute()` is called to strip away all the non-metal information from `wiregrid[][]`. The procedure `makewire()` is called to walk through the newly generated wire and fill the appropriate layout arrays, generating `rect` structures that correspond to the wire. At this point `printgrid()` can be called to dump the new wire into the `metalgrid[][]` array, and routing of a new wire can begin.

**Route Checking**

Once a cell is issued from the route queue its next possible moves need to be checked to see if another valid move can be made, which is accomplished by the TLG `routecheck()` procedure. This procedure entails, for the most part, a brute force method of making metal width and metal spacing considerations (or polysilicon considerations). Depending on the current material layer used for routing, a new candidate is first checked to ensure a wire would be able to fit within the allotted space. Note that all of these procedures make good use of the bit masking technique alluded to earlier. In the second stage of the checking process, the space around the wire is checked to make sure it does not interfere with surrounding oxides, diffusion contacts, other wires, or other obstacles. This check was initially accomplished by extending the current wire an appropriate distance up, down, left, and right, but it was discovered that this did not account for interference on the "diagonals" of the routed wire. The checking process was updated to reflect this change, and it is depicted in **Figure 11** below.

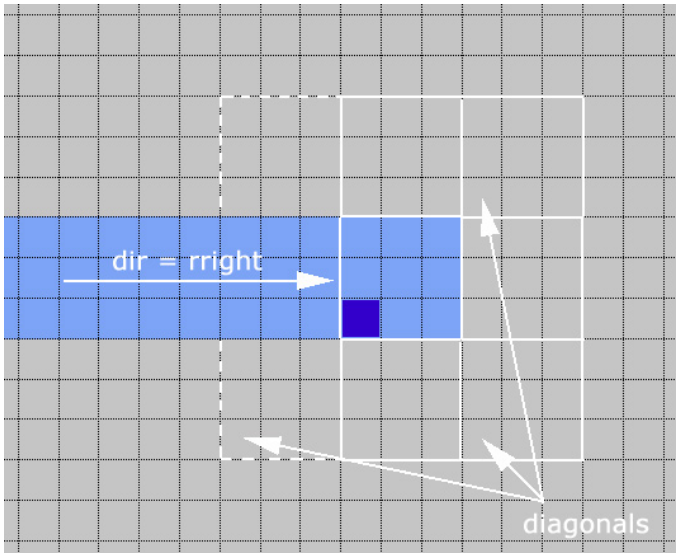



**Figure 11:** Wire Spacing Checking

The second aspect of route checking (though not explicitly in the `routecheck()` procedure) is the method used to identify if a cell is approaching its destination. As one might imagine, if the checking procedure is implemented too aggressively a wire approaching its target may interpret the destination as an obstacle that needs to be avoided. In this scenario no routing would ever complete successfully. To avoid this dilemma the concept of a destination "halo" is introduced. A destination halo encompasses the destination contact (or polysilicon) itself, as well as any obstacles that might be within the reach of the halo. Thus, if a cell approaching a destination encounters an obstacle within the destination halo, the obstacle is assumed to be electrically connected to the destination. At this point it is acceptable to connect the wire to the obstacle instead of the destination.

### Multilayer Routing

Multilayer routing is in many ways just an extension of wire routing. It is, in essence, the same routing algorithm applied in three dimensions instead of two. However, enough changes need to be made to the algorithm and checking routine that warrant discussion.

To make the TLG wire routing procedure work on multiple metal layers, the two-dimensional array `wiregrid[][] is` made three-dimensional (while `metalgrid[][]`

remains the same, however). The assignment of metal layers to layers of the grid can depend on the user, but using `wiregrid[0][x][y]` for the polysilicon layer, `wiregrid[1][x][y]` for metal layer 1, `wiregrid[2][x][y]` for metal 2, etc. is generally a good idea. It is a particularly good idea to keep the `wiregrid` layers in the same order as their physical counterparts because it makes moving up and down metal layers much more intuitive. For example, one counter can simply be incremented or decremented when changing layers. The implication of adding an extra dimension to the array is that all accesses to and from `wiregrid[][][]` are layer specific, and metal layer information needs to be carried around in nearly all of the wire routing procedures. Additionally, as was noted earlier, metal information is added to the `cell` structure itself to facilitate multilayer functionality.

The TLG routing and retracing functions are generalized to accommodate transitions between metal layers and different wire lengths. Forward routing is not as much of a problem, because the purpose is still to get to the destination, and having misalignment in the wiregrid while moving forward is not a huge concern. Additional statements in routewire() allow cells to move both "in" and "out" of `wiregrid[][]`[] (down and up metal layers, respectively). Backward retracing, however, becomes a major pain to get working correctly because cell blocks are no longer guaranteed to be aligned. It is perfectly conceivable, for example, for a wire to hop down from metal 1 to polysilicon, travel a bit, then hop back up to metal 1. Since the metal 1 width and polysilicon widths are generally different the metal 1 layer will no longer be aligned to metal 1 strides. This issue is resolved by adding a "crawling" mechanism to the retracewire() procedure. On every iteration into retracewire(), the procedure crawls as far as it can to the lower left and upper right corners of the current cell, provided the count over all the points within the cell remains constant. The procedure then fills in this entire region, and looks for the next region to move to by searching around the entire perimeter of the current cell, and the entire slice of area directly above and below the current cell. Using this method is effective in finding a path back to the source node, but the misalignment issues can occasionally leave extra bits of material hanging off the ends of the wires. Altogether

these issues give further support for a change to single-lambda strides in future versions of TLG.

Further enhancements to the TLG `routecheck`() procedure are made to check if moves between material layers can be made. Not only does the space above (or below) the cell need to be clear, but room needs to be made for a via (or polysilicon contact) to join the two layers together. Additionally, the contact needs to be spaced away from obstacles according to the design rules specified. This is especially annoying since polysilicon contacts can really mess around with the generated wires, but it is necessary for functional layout. Lastly, the final `makewire()` procedure is altered to generate `rect` layout for all metal layers and vias that a new multilayer wire requires.

Once these modifications are made to TLG, source-to-gate and drain-to-gate wire routing is possible. Additionally, previous connections that may have been unreachable might find ways to become connected once other metal layers are usable. It is also possible to specify certain metal layers as completely unusable, or define a severe penalty if they are used by modifying the distance count with a weight when moving between material layers. Also, it is possible to specify an orientation preference for metal layers, such as "metal 1 goes vertically, metal 2 goes horizontally". Not all of these options are immediately available to the user in this version of TLG, but the infrastructure is completely in place to make it possible.

# 6. Future Work

Throughout this document a number of issues have been highlighted as areas that TLG can improve upon for future versions. It is truly hoped that other students are inspired by this work, and get the opportunity to continue the development of this potentially invaluable tool. Some future work on TLG could include some of the improvements or features described in the sections below.

### Improvements

First, the `trans` structure, while full of many bells and whistles might be greatly simplified if transistors we interpreted to be completely symmetric; namely if source and drain connections were not explicitly specified in the structure. This alteration would, among other things, greatly simplify the `transfix()` procedure, and in general clean up the entire stack generation process. Second, the current method of substrate contact generation is sufficient for small designs, but may be inadequate for larger ones. More tests will be able to tell if this is indeed the case, and if a more robust version needs to be implemented. Third, it is clear that using single-lambda strides while wire routing is a needed feature, both to remove all alignment issues, and to improve the chances that a wire can be routed within a tight space.

Additional improvements may include a volleying back and forth between the stack generation process and the wire routing phases of TLG to optimize the generated layout. While stacks that are currently generated do preemptively adjust their positions in anticipation of wiring, once they are generated they are fixed. There are choices that are made in determining where a stack should begin and end that may adversely impact the ability of the stacks to be wired effectively, and this issue is not addressed by the current version of TLG. Additionally, stack positions could be adjusted relative to each other to better align gate connections or create full-cell-length busses instead of point-to-point wires. Also, the overall wiring efficiency in general could be improved by analyzing total wire lengths, or some other sufficient cost metric, to determine if certain wires should be re-routed to improve the "common good" of the generated layout.

### New Features

As mentioned before TLG can be expanded in nearly every direction. This is the type of project that has no real end; it is like a painting that has no clear final brush stroke. There are countless new features that could be added to TLG. If the arraying of single cell layout blocks were added (with possible bit-pitch considerations) TLG would be one step closer to being able to generate a full microprocessor. Among other alterations TLG

would need to generate hierarchical "uses" information in the resulting `mag` file. If a parser was created for inputting `mag` files, TLG could be adapted to wire existing MAGIC layout files. This could be especially useful, but it would be very tricky to store all the needed information compactly. TLG could be updated to receive input directly from technology files. This would be an excellent feature because they would not need to be encoded in the top of the `tlg` files anymore.

Additional possible features include providing support for busses, ensuring all nodes of a cell are available on the perimeter of the cell, and allowing the user to "rubber stamp" layouts by typing in a word or phrase and having it automatically generate the layout for them. These are merely a few ideas that can be implemented on the existing TLG framework. I look forward to seeing what else can be done.

# 7. Results and Conclusions

Overall TLG performs quite admirably in automatically generating all kinds of single-cell transistor layout, ranging from incredibly simple designs (one or two transistors) to some relatively tricky configurations. A collection of snapshots of layout generated by TLG in response to particular input files is included in **Appendix E**. While not particularly impressive layout, it is completely functional, free of design rule violations, and usually has no extract warnings or wellcheck errors. Occasionally connections are deemed inaccessible when a clever designer would probably be able to manually fit a wire there, but these instances are forgivable. A number of unique data structures and algorithms were created for the explicit purpose of automated transistor layout generation, and C proved to be a perfectly acceptable language to develop in. The results for each phase of TLG's execution in light of the specifications and expectations of **Table 1** are discussed in the following sections.

### TLG Operation

TLG operates exactly how it was intended to: it takes in an input file, performs stack generation and wire routing operations on it, and generates a fully MAGIC compliant

output file. While TLG does not read in CAST files directly, it reads in its own formatted `tlg` files which should be easy to generate from CAST files. TLG clearly is capable of successfully generating basic transistors with labels, and parsing transistor characteristics input from the `tlg` file to alter the layout. Also, TLG is able to parse definitions from specified lines in the `tlg` file to modify design rules and the general behavior of TLG's execution.

**Stack Generation**

TLG is fully capable of generating connection graphs from input transistor parameters. From these graphs it is able to successfully generate nfet and pfet transistor stacks, sharing source or drain connections accordingly. In generating these stacks, additional considerations are given to nodes that are power rails or output nodes, and diffusion contacts, labels, and substrate contacts are inserted into the layout as needed. The substrate contact insertion is not particularly robust and could use improvement, but the feature was never a high priority in the scope of the entire project. Lastly, loop stacks are treated just as effectively as "normal" transistor stacks.

**Wire Routing**

Getting TLG to perform wire routing effectively proved to be a very challenging task, but after much struggling it was finally accomplished. The stack structures are correctly output to grids, needed connections are positively identified, and requests for wires are placed into a route queue. The Maze Routing algorithm is successfully used to find an optimal path for routing a particular wire, and design rule considerations of metal width and spacing are incorporated to ensure correct wiring practices. Occasionally it is obvious that a previously generated wire will become a problem for future wires but there is no current method implemented to deal with this issue.

Once the destination is found by TLG the wire was retraced with an effort to keep the wire as straight as possible. In the case of multilayer routing, the retraced wire would occasionally have jagged edges as a result of cell misalignment. Placing vias and

contacts down when switching metal layers proved to be a bit tricky, but ultimately posed no major problem. Infrastructure to enable constraints on the metal layers is in place, but has not been fully implemented.

The single-layer and multilayer wire routing algorithms have been tested, but by no means have they been tested exhaustively, and it is expected that certain transistor configurations will give TLG some significant problems. In fact, it has already been observed that TLG will occasionally cause segmentation faults when running a sufficiently complex transistor configuration. This error could be a result of inefficient usage of data structures, or a coding error, but is more likely a byproduct of the Maze Routing algorithm itself which requires a lot of memory to find a sufficiently far destination node. It is hoped that this version of TLG will be "Beta" tested by undergraduate students or M.Eng./Ph.D. students to see if it is truly valid, hence the term "Beta" in the title of the report. Incidentally, the 0.7 in the title arises because even though TLG is quite complete in its own right, without a proper parser of CAST files, and other miscellaneous bugs fixed it does not seem like a complete version 1.0. Additionally, 0.7 volts is a standard voltage drop across the base-emitter junction of a BJT :-).

**Final Thoughts**

This project is definitely a success. Not only have I successfully constructed a complete program that is capable of automatically generating layout from production rules, I have learned an incredible amount about automated stack generation, wire routing, and optimization techniques. Above all I have an even better appreciation for all the time that was spent laying out the tic-tac-toe microprocessor [5] I helped develop in ECE 474.

For every solution it always seems there are three others that are worth a try. For many of the design decisions that were made, there are other possible implementations that may produce better results. However, it is somewhat reassuring to know there are also undoubtedly many implementations that will produce far worse results :-).

Judging by the length of this report alone it would seem that I'd be able to "write a book" about this topic. Even so, it is unfortunately impossible for me to document every single clever implementation that was used in this project. However, since it is a subject that interests me so greatly, and the development of TLG has given me an excellent opportunity to innovate, writing a book about the topic is something I might actually consider someday.

So in closing, one may ask: does TLG allow for concurrent generation of a complete microprocessor? Not yet. Unless you find a good way to split a microprocessor up into `tlg` files and merge the resulting layouts, this version of TLG can't quite get you there. Will it save students and other VLSI layout designers time? Quite possibly. TLG generates acceptable layout just frequently enough for it to be considered useful. With improved optimizations, the ironing out of a few bugs, and the addition of some key features that future versions might bring, TLG may prove to be an invaluable tool for Cornell ECE students performing VLSI design. At least I sincerely hope it does.

## 8. Acknowledgments

Many thanks to Professor Rajit Manohar for all of his many insights into VLSI layout, C coding techniques, and life in general. It has been my distinct pleasure working with him and all of the ECE faculty and students and I have encountered over the past five years.

## 9. References

[1] "CAST and IRSIM". http://vlsi.cornell.edu/courses/ece474/cast.html. Accessed: May 2003.

[2] "Magic – A VLSI Layout System". http://vlsi.cornell.edu/magic/. Accessed: May 2003.

[3] "Magic – Format of .mag Files Read/Written by Magic". Accessed: May 2003.

[4] John A. Nestor. "Multilayer Maze Routing Demonstration Applet" http://foghorn.cadlab.lafayette.edu/cadapplets/MultiMaze.html. July 27th, 2001. Accessed: May 2003.

[5] Victor Aprea, Paul Grzymkowski, and Andre Kozaczka. "3T: Innovation in Tic-Tac-Toe." Fall 2001 (unpublished)

# 10. Appendices

**Appendix A:** TLG Code Summary

```
// position information
int orgx, orgy, topx, topy;

// definitions
int dh, pw, ppw, m1w, m1m1w, m2w, m2m2w, ndpdh, ndch, ndndch, pdch,
      pdpdch, pcw, pch, v1w, v1h, v2w, v2h, minl, minw;

// layout arrays
rect *ndiff, *pdiff, *poly, *ndc, *pdc, *m1, *m2, *pc, *via1, *via2,
      *psc, *nsc;
label *labels;

// layout array counters
int ndcount, ndccount, nsccount;
int pdcount, pdccount, psccount;
int fcount, polycount, lcount, scount, wcount, m1count, m2count,
      pccount, v1count, v2count;
int lm1count, lm2count, lpccount;

// i/o variables
char filename[linesize], outname[linesize];
char line[linesize];

char tlgexit;
char stackposition, stackgen, wirerouting, gaterouting, multilayer,
      wellplugs, verbose, quiet;

// stack generation variables
trans *fets;
strans *stacks;

// wire routing variables
wire *wires;
int **metalgrid, ***wiregrid;
char routefound, routecount, lastroutecount;
int routem;
int foundtx, foundty;
cell *routequeue, *routetail;
int neededwires, foundwires;

// rect struct - stores layout rectangles
struct rect {int xbot; int ybot; int xtop; int ytop;};

// label struct - stores layout labels
struct label {char *layer; rect r; int pos; char *name;};

// trans struct - defines transistors and interconnections while
// reading .tlg file. walked and modified during stack generation
struct trans {int l; int w; char s[labsize]; struct trans *sleft;
      struct trans *sright; char d[labsize]; struct trans *dleft;
      struct trans *dright; char g[labsize]; struct trans *gleft;
```

```
      struct trans *gright; struct rect r; int sout; int dout; int
      gout; int color; char type;};

// wire struct - stores wire request information
struct wire {int x; int y; int s; char *name; int color;};

// cell struct - represent new areas of wiregrid to visit
struct cell {int x; int y; int tx; int ty; int m; int tm; int count;
      struct cell *next; char dir;};

// strans struct - represent transistor stacks
struct strans {struct trans *t; int color; struct strans *left; struct
      strans *right;};

/* getmeat - read input from .tlg, create transistors, and link them
      using locate */
void getmeat()

/* locate - look through existing transistors and link to new one if
      node names match */
void locate(trans *fet, char *name, int sdg)

/* printstack - try to find stack heads, use stackwalk to find rest */
void printstack()

/* stackwalk - walk through transistor connections until have to stop
      */
void stackwalk(trans *t, char *name)

/* transfix - remove a transistor and tie the remaining connections
      together */
void transfix(struct trans *t)

/* makestack - walk over stacks and generate rects for layout arrays
 *           - add diffusion contacts and labels
 *           - add substrate plugs if desired
 *           - identify connections that still need wires
 */
void makestack()

/* printgrid - dump current contents of layout arrays to metalgrid */
void printgrid()

/* gridfill - fill a rectangle of metalgrid with a value */
void gridfill(rect r, int val)

/* routegrid - service wire requests by finding a connetion point and
      adding cells to be explored */
void routegrid()

/* routewire - check if new spaces are available and add cell requests,
      or the destination has been reached */
char routewire(int sx, int sy, int tx, int ty, int count, char dir, int
      m, int tm)

/* routecheck - brute force check if current move is valid according to
      metalgrid and wiregrid */
```

```
char routecheck(int x, int y, int tx, int ty, char dir, int m, int mp,
      int tm)

/* addcell - allocate memory for a new cell and add it to the route
      queue */
void addcell(int x, int y, int tx, int ty, int count, char dir, int m,
      int tm)

/* routefill - fill the desired square of wiregrid with current
      distance count */
void routefill(int x, int y, int count, char dir, int m)

/* emptyqueue - service the cell requests by running routewire */
void emptyqueue()

/* retracewire - trace the wire back to the source following the
      distance counters */
char retracewire(int x, int y, int count, char dir, int m)

/* clearroute - clear away everything from wiregrid except the
      generated wire */
void clearroute()

/* travwire - traverse the wire to find successful connections */
void travwire(int x, int y, int dx, int dy, int count, char dir, int m,
      int tm)

/* makewire - create wire layout data from rectangles in wiregrid */
void makewire()

/* makehead - write header to .mag file */
void makehead()

/* makemeat - output the layout arrays to the .mag file */
void makemeat()

/* maketail - write footer to .mag file */
void maketail()

/* makerect - add a layout rectangle to .mag file */
void makerect(rect r, char *s)

/* makelabel - add a layout label to the .mag file */
void makelabel(label l, char *s)
```

**Appendix B:** TLG File Format Definitions

| Definition | Description | Default Value |
|---|---|---|
| **dh:** | min. diffusion height | 3 |
| **pw:** | polysilicon width | 2 |
| **ppw:** | min. poly-poly distance | 3 |
| **m1w:** | metal 1 width | 3 |
| **m1m1w:** | metal1-metal1 distance | 3 |
| **m2w:** | metal 2 width | 3 |
| **m2m2w:** | metal2-metal2 distance | 3 |
| **ndpdh:** | n-diffusion to p-diffusion height | 12 |
| **ndch:** | n-diffusion contact size | 4 |
| **ndndch:** | n-diff to n-diff contact buffer | 1 |
| **pdch:** | p-diffusion contact size | 4 |
| **pdpdch:** | p-diff to p-diff contact buffer | 1 |
| **pcw:** | poly contact size | 4 |
| **pch:** | poly contact halo size | 4 |
| **v1w:** | via 1 size | 4 |
| **v1h:** | via 1 halo size | 3 |
| **v2w:** | via 2 size | 4 |
| **v2h:** | via 2 halo size | 3 |
| **minl:** | min. transistor length | 2 |
| **minw:** | min. transistor width | 5 |

**Appendix C:** MAG File Available Layers

```
<< ntransistor >>
<< ptransistor >>
<< ndiffusion >>
<< pdiffusion >>
<< ndcontact >>
<< pdcontact >>
<< psubstratepcontact >>
<< nsubstratencontact >>
<< polysilicon >>
<< polycontact >>
<< metal1 >>
<< m2contact >>
<< metal2 >>
<< m3contact >>
<< metal3 >>
<< labels >>
```

## **Appendix D:** Sample .wire and .grid Outputs

### Sample `metalgrid[][]` Output:

`threetrans.grid`

```
00000000000000000000000                                  00000000000000000000000
00000000000000000000000                                  00000000000000000000000
00000000000000000000000                                  0000EEE0000000000000000
00000000000000000000000                                  0000EEE0000000000000000
00000000000000000000000                                  0000EEE0000000000000000
00000000000000000000000                                  0000DDDEEE0000000DDD000
00000000004444400000000                                  0000DDDEEE0000000DDD000
00000000004444400000000                                  0000DDDEEE0000000DDD000
00000000004444400000000                                  0000CCCDDDEEE0DDDCCC000
00000000004444400000000                                  0000CCCDDDEEE0DDDCCC000
00000000004444400000000                                  0000CCCDDDEEE0DDDCCC000
00000001133333311000000                                  0000BBBCCCDDDDCCCBBB000
00000001133333311000000                                  0000BBBCCCDDDDCCCBBB000
00000000004444400000000                                  0000BBBCCCDDDDCCCBBB000
00000000004444400000000                                  0000AAA0000000000AAA000
00000000004444400000000                                  0000AAA0000000000AAA000
00000000004444400000000                                  0000AAA0000000000AAA000
00000000004444400000000                                  0000999000000000999000
00000000004444400000000                                  0000999000000000999000
00000001133333311000000                                  0000999000000000999000
00000001133333311000000                                  0000888000000000888000
00000000004444400000000                                  0000888000000000888000
00000000004444400000000                                  0000888000000000888000
00000000004444400000000                                  0000777666555566777000
00000000004444400000000                                  0000777666555566777000
00000000004444400000000                                  0000777666555566777000
00000000004444400000000                                  0000888000000000888000
00000001133333311000000                                  0000888000000000888000
00000001133333311000000                                  0000888000000000888000
00000000004444400000000                                  0000999000000000999000
00000000004444400000000                                  0000999000000000999000
00000000004444400000000                                  0000999000000000999000
00000000004444400000000                                  0000AAA0000000000AAA000
00000000004444400000000                                  0000AAA0000000000AAA000
00000000000000000000000                                  0000AAA0000000000AAA000
00000000000000000000000                                  0000BBB0000000000BBB000
00000000000000000000000                                  0000BBB0000000000BBB000
00000000000000000000000                                  0000BBB0000000000BBB000
00000000000000000000000                                  0000CCCDDD0000DDDCCC000
00000000000000000000000                                  0000CCCDDD0000DDDCCC000
00000000000000000000000                                  0000CCCDDD0000DDDCCC000
00000000000000000000000                                  00000000000000000000000
00000000000000000000000                                  00000000000000000000000
00000000000000000000000                                  00000000000000000000000
012345678901234567890123
```

### Sample `Wiregrid[][]` Output:

`threetrans.wire`

`(note: before the call to retracewire)`

## **Appendix E:** Sample TLG Outputs

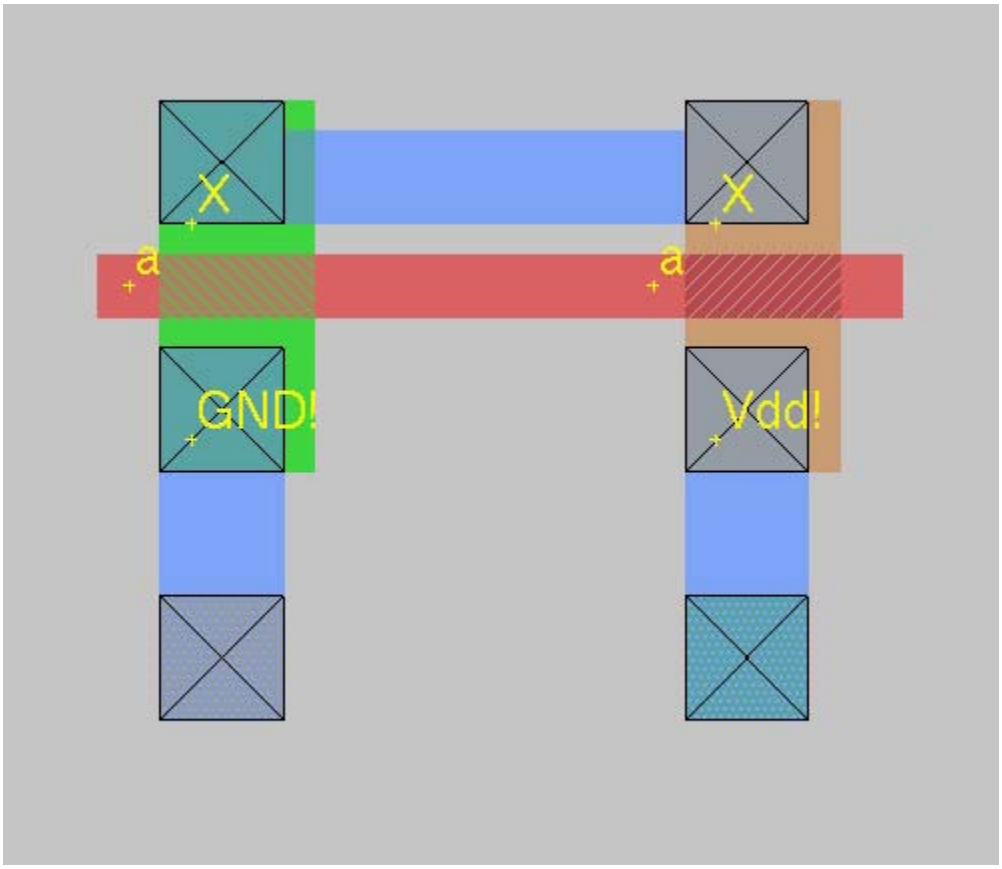
**TLG File:** `inv.tlg`

```
| inv.tlg - a simple inverter |
nfet l:2 w:5 s:GND! g:a d:X
pfet s:Vdd! g:a d:X
```

**TLG Command:**

```
$ tlg inv -swgc
```

**MAG Output:**



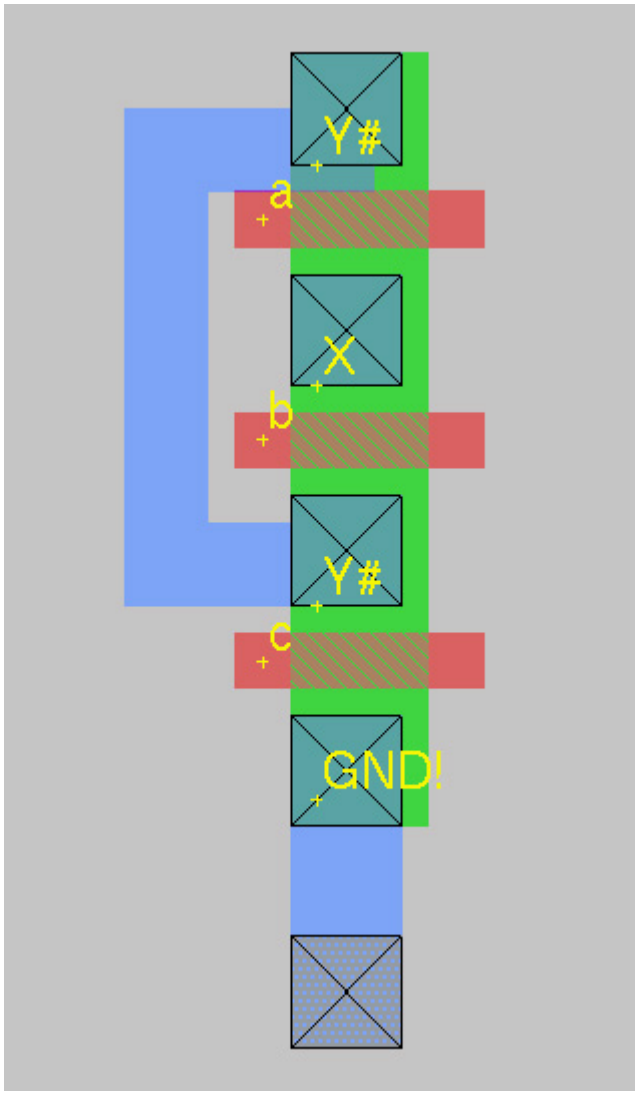
**TLG File:** `threetrans.tlg`

```
| threetrans.tlg - 3 fets |
nfet s:GND! d:Y# g:c
nfet s:Y# d:X g:b
nfet s:Y# d:X g:a
```

**TLG Command:**

```
$ tlg threetrans -swgc
```

**MAG Output:**



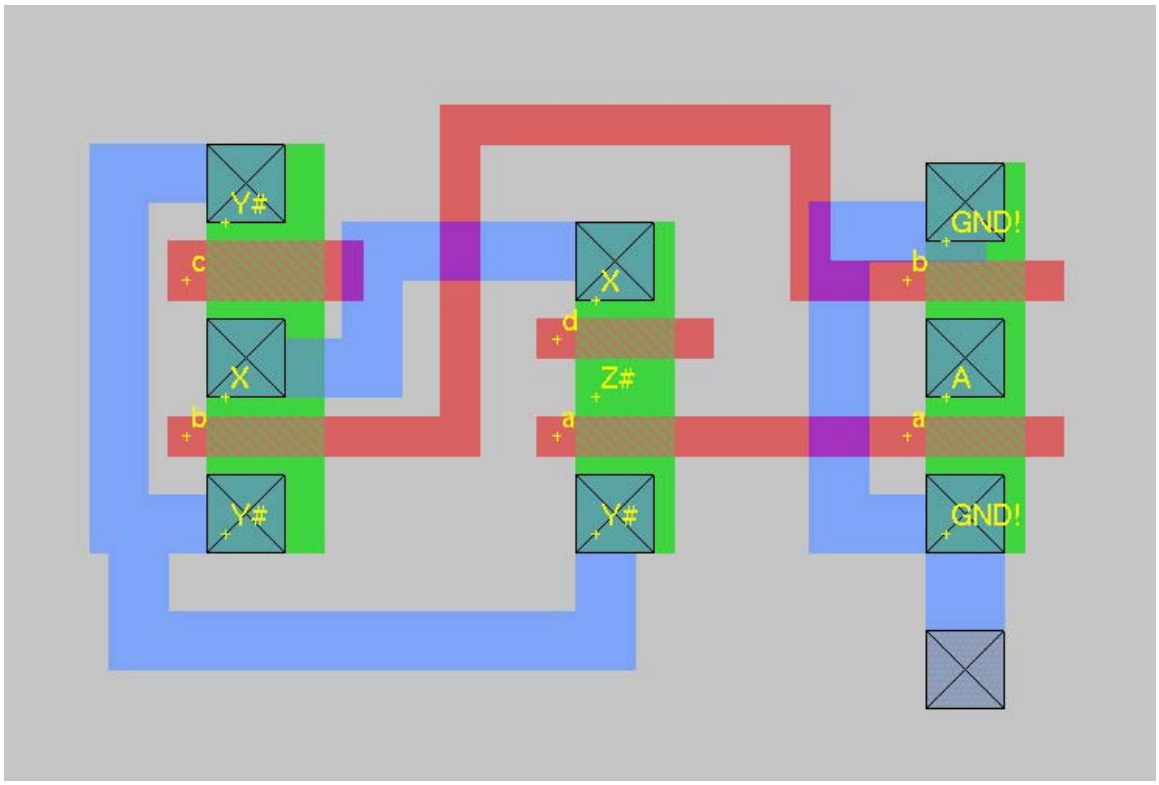
**TLG File:** `route.tlg`

```
| nfet stack
nfet w:6 s:Y# d:X g:b
nfet l:3 s:X d:Y# g:c
nfet s:Y# d:Z# g:a
nfet s:Z# d:X g:d

nfet s:GND! d:A g:a
nfet s:GND! d:A g:b
```

**TLG Command:**

```
$ tlg route -swgc
```

**MAG Output:**



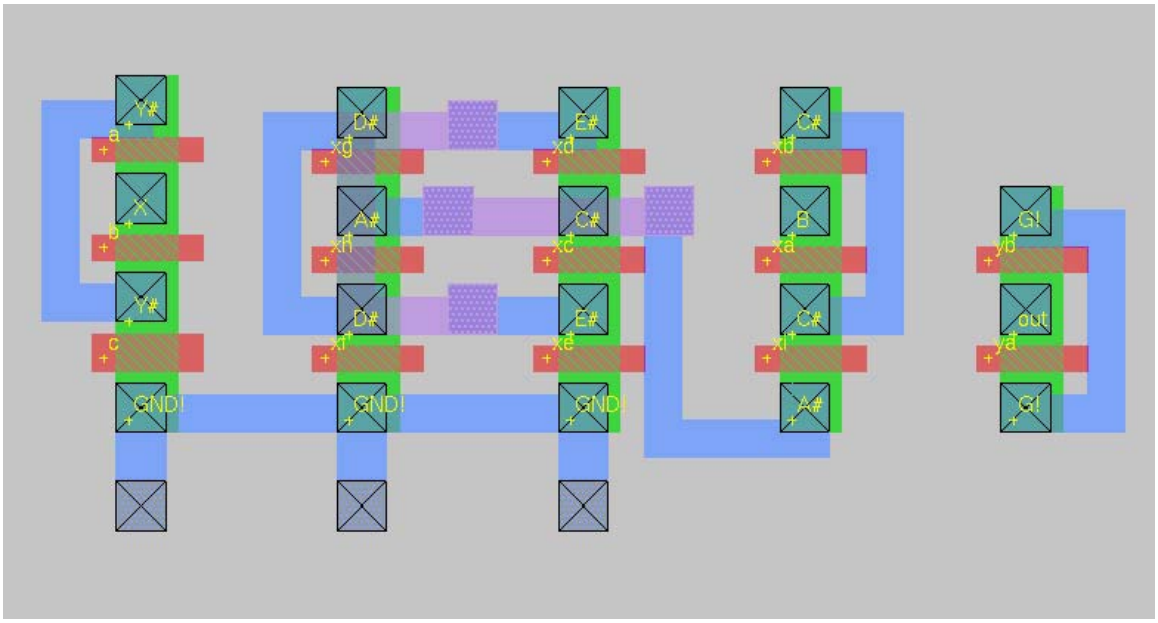
**TLG File:** `stacks.tlg`

```
| nfet stack 1
nfet l:3 s:GND! d:Y# g:c
nfet w:6 s:Y# d:X g:b
nfet s:Y# d:X g:a

| nfet stack 2
nfet s:GND! d:D# g:xf
nfet s:GND! d:E# g:xe
nfet s:D# d:A# g:xh
nfet s:D# d:A# g:xg
nfet s:E# d:C# g:xc
nfet s:E# d:C# g:xd
nfet s:A# d:C# g:xi
nfet s:C# d:B g:xa
nfet s:C# d:B g:xb

| nfet stack 3
nfet s:G! d:out g:ya
nfet s:G! d:out g:yb
```

**TLG Command:**

```
$ tlg stacks -uc
```

**MAG Output:**



**TLG File:** `lab1.tlg`

```
|| ECE 474 - Lab 1
|| lab1.tlg - three inverters

pfet s:Vdd! g:i1 d:o1
nfet s:GND! g:i1 d:o1

pfet s:Vdd! g:o1 d:o2
nfet s:GND! g:o1 d:o2

pfet s:Vdd! g:o2 d:o3
nfet s:GND! g:o2 d:o3
```

**TLG Command:**

```
$ tlg lab1 -uv
```

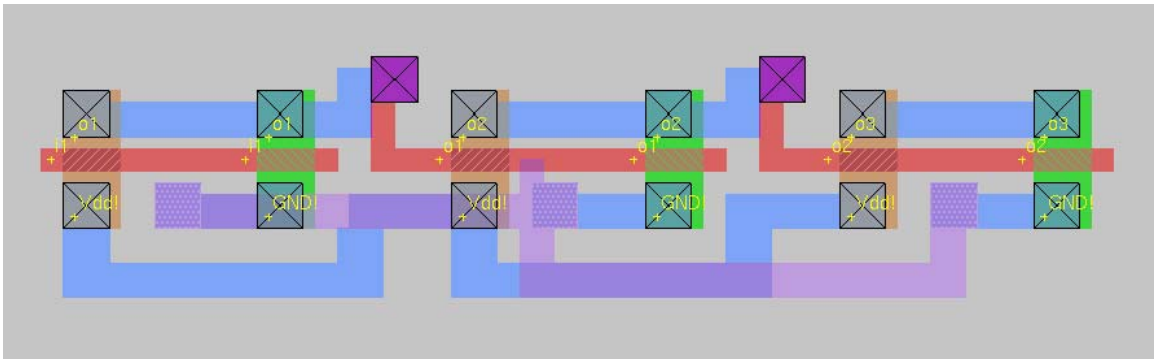
**TLG Output:**

```
Note: no output file specified, lab1.mag assumed
Input file=lab1.tlg
Output file=lab1.mag

TLG: Generating stacks...

pfet Created! Source=Vdd! Drain=o1 Gate=i1
nfet Created! Source=GND! Drain=o1 Gate=i1
pfet Created! Source=Vdd! Drain=o2 Gate=o1
nfet Created! Source=GND! Drain=o2 Gate=o1
pfet Created! Source=Vdd! Drain=o3 Gate=o2
nfet Created! Source=GND! Drain=o3 Gate=o2

Vdd!:3-(i1:2)-o1:4
GND!:3-(i1:2)-o1:4
Vdd!:3-(o1:4)-o2:4
GND!:3-(o1:4)-o2:4
Vdd!:3-(o2:4)-o3:2
GND!:3-(o2:4)-o3:2

Vdd! (9, 10) needs a wire:0
o1 (9, 18) needs a wire:1
i1 (7, 15) needs a wire:2
GND! (26, 10) needs a wire:3
o1 (26, 18) needs a wire:4
i1 (24, 15) needs a wire:5
Vdd! (43, 10) needs a wire:6
o2 (43, 18) needs a wire:7
o1 (41, 15) needs a wire:8
GND! (60, 10) needs a wire:9
o2 (60, 18) needs a wire:10
o1 (58, 15) needs a wire:11
Vdd! (77, 10) needs a wire:12
o3 (77, 18) needs a wire:13
o2 (75, 15) needs a wire:14
GND! (94, 10) needs a wire:15
```

```
o3 (94, 18) needs a wire:16
o2 (92, 15) needs a wire:17

TLG: Routing wires...

Vdd!  (9,10) to (43, 10)  ROUTE FOUND (28)!
o1    (9,18) to (26, 18)  ROUTE FOUND (10)!
i1    (7,15) to (24, 15)  ROUTE FOUND (10)!
GND!  (26,10) to (60, 10)  ROUTE FOUND (34)!
o1    (26,18) to (41, 15)  ROUTE FOUND (20)!
Vdd!  (43,10) to (77, 10)  ROUTE FOUND (20)!
o2    (43,18) to (60, 18)  ROUTE FOUND (10)!
o1    (41,15) to (58, 15)  ROUTE FOUND (10)!
GND!  (60,10) to (94, 10)
o2    (60,18) to (75, 15)  ROUTE FOUND (20)!
o3    (77,18) to (94, 18)  ROUTE FOUND (10)!
o2    (75,15) to (92, 15)  ROUTE FOUND (10)!
GND!  (94,10) to (60, 10)
      (94,10) to w(60, 10)
      (94,10) to w(59, 10)
      (94,10) to w(58, 10)
      (94,10) to w(57, 10)
      (94,10) to w(56, 10)
      (94,10) to w(55, 10)
      (94,10) to w(54, 10)
      (94,10) to w(53, 10)
      (94,10) to w(52, 10)
      (94,10) to w(51, 10)
      (94,10) to w(50, 10)
      (94,10) to w(50, 11)
      (94,10) to w(51, 11)
      (94,10) to w(52, 11)
      (94,10) to w(53, 11)
      (94,10) to w(54, 11)
      (94,10) to w(55, 11)
      (94,10) to w(56, 11)
      (94,10) to w(57, 11)
      (94,10) to w(58, 11)
      (94,10) to w(59, 11)
      (94,10) to w(60, 11)
      (94,10) to w(61, 11)
      (94,10) to w(61, 10)
      (94,10) to w(61, 12)
      (94,10) to w(60, 12)
      (94,10) to w(59, 12)
      (94,10) to w(58, 12)
      (94,10) to w(57, 12)
      (94,10) to w(56, 12)
      (94,10) to w(55, 12)
      (94,10) to w(54, 12)
      (94,10) to w(53, 12)
      (94,10) to w(52, 12)
      (94,10) to w(51, 12)
      (94,10) to w(50, 12)
      (94,10) to w(50, 12)  ROUTE FOUND (31)!
```

```
TLG: lab1.mag generated successfully.
```

**MAG Output:**



## **Appendix F:** TLG Operation Guide

### TLG Execution

TLG can be executed by issuing following command:

```
$ tlg filename [-spwgmcuovq] [position] [outputfile]
```

The only required parameter into TLG is the input tlg file. It is not necessary to include the .tlg extension on the command line, but the actual filename must contain the .tlg extension. From here the following operators can optionally be included in the command line:

**TLG Runtime Parameters**

| Operator | Description |
|---|---|
| **-s** | Stack gen. enable |
| **-p** | Stack positioning |
| **-w** | Wire routing enable |
| **-g** | Gate routing enable |
| **-m** | Multilayer routing |
| **-c** | Well plugging enable |
| **-u** | "Usual" configuration |
| **-o** | Output specification |
| **-v** | Verbose mode |
| **-q** | Quiet mode |

The operators –s, -w, -g, -m, and –c enable transistor stacking, wire routing, gate routing, multilayer routing, and well plugging, respectively. Additionally enabling the operator –u has the same effect as using the operators –s, -w, -g, and –m, signifying the "usual" TLG configuration. Note that if no operators are specified at all the "usual" TLG mode is assumed.

The operators –v and –q stand for "verbose" mode and "quiet" mode respectively. Execution in "verbose" mode will output all intermediate procedural information to stdio, while "quiet" will suppress all messages other than errors. Note further that "quiet" takes precedence over "verbose" mode.

The operator –p enables stack position preference. Using this flag requires an additional input `position` after the operators with a "0" representing in-order stacking, "1" representing horizontal nfet and pfet grouping, and "2" representing vertical nfet and pfet grouping. The operator –o enables the output filename to be specified. After using this flag the desired filename `outputfile` is required after the operators. Note that for an input file "file.tlg" the default output is "file.mag" if no output filename is specified. Note that the order of the additional `position` and `outputfile` parameters is dependent on the order of the –p and –o operators.