CMOS CALCULATOR

Joshua Clark

Third Year Project Report for the Final
Honour School of Computer Science

May 2015

Abstract

The aim of this project was to build a CMOS simulator, with the following interactive elements:

- Simplification and CMOS implementation of arbitrary logical expressions
- Simulation to visualise the change in flow of potential as the state of inputs changed
- Simulation of delay in propagation of state for large networks.

There are many tools in existence to design CMOS circuits, but most focus on the physical implementation and the exact positioning of transistors, rather than a more abstract approach, as may be required when reasoning about the layout of circuits, this tool seeks to remedy that issue. It quickly became apparent that the approach that I was using had various similarities to a compiler, which will be a focus of this document. Additionally, this report will discuss alternative approaches not used in the final implementation to produce the final CMOS output.

Contents

List of Algorithms

List of Figures

1 Introduction

The idea of logical functions is familiar to many, and even a schoolchild can grasp the concept of a box receiving true/false inputs and producing a true/false output, and lots of these boxes in interesting combinations can yield more interesting functions. Through introducing these concepts, and producing more and more intricate diagrams, they can then see how more complicated ideas, such as addition and subtraction are implemented, and maybe how these are used in the modern computers used today. However, what is rarely discussed is what is inside these boxes: they are usually presented as the lowest level inputs and treated as the basic building blocks. The layer underneath this is built up of Complementary Metal-Oxide Semiconductors, henceforth referred to as CMOS. The treatment of this document mainly concerns producing this sequence of transistors from plain string through a series of intermediary formats, until a visual representation is produced.

Initially, this document will define key concepts, such as the fundamental propositional logic syntax and semantic meanings usually associated with logic gates, and the definition of the two different transistors that form CMOS. We will then go on to discuss the requirements and use cases of a program to present these ideas before defining the algorithms to be used. On closer inspection, their composition has a behaviour not entirely dissimilar to that of a compiler, and the subsequent design of the program upon realising this was the paradigm being used will then be analysed. The program that has then been produced will finally be compared against the stated objectives, with possible further work then listed.

1.1 Document Conventions

By convention, when class diagrams are used, ovals will represent traits/abstract classes, with rectangles representing concrete implementations. Objects are indicated using bold lines in the diagram, and are similar in behaviour to statically declared objects in many languages.

Whenever `this font face` is used, the text is referring to either some implementation level object or class, or the grammar used by the parser as opposed to the standard blackboard mathematical symbols, or text entered into the application or used for testing.

1.2 Definitions

1.2.1 Logical Expressions

A logical expression is built from a combination of constants, literals, conjunctions, disjunctions and negations[9], using a grammar described as follows:

$$
\begin{aligned}
e = \neg e = {} & \texttt{!e} && (1)\\
\mid e \vee e = {} & \texttt{e or e} && (2)\\
\mid e \wedge e = {} & \texttt{e and e} && (3)\\
\mid c & && (4)\\
\mid v & && (5)\\
v \in \Sigma & && (6)\\
c = \top \mid \bot = {} & \texttt{0} \mid \texttt{1} && (7)
\end{aligned}
$$

where Σ is some accepted alphabet of variables (in this program, as described by the Perl-Compatible Regex `^(?!out|and|or$)([A-Za-z0-9]+)` that is, every alphanumeric phrase, with the exception of the single word `out`, which has been reserved for specific use within the program, and `and` and `or` which are keywords in the logical expression language). $\top$ and $\bot$ are used within this text to denote the notions true and false, respectively, and are represented in the program by the strings `0` and `1` respectively. Additionally, the order that the various production rules are listed also gives an order of precedence, with negations binding more tightly than disjunctions and conjunctions. Each variable within the expression that is used is given an assignment by an evaluation function, commonly called $\mathcal{A} : \mathcal{V} \to \{\top, \bot\}$, with

the set of variables typically denoted as $\mathcal{V}$. A logical expression E is then evaluated recursively based on the following rules:

- $E = F \wedge G$: E is evaluated as true iff F is true and G is true (corresponding to equation 1)
- $E = F \vee G$: E is evaluated as true iff F is true or G is true (corresponding to equation 2)
- $E = \neg F$: E is evaluated as true iff F is false (corresponding to equation 3)
- $E = v$ for some $v \in \Sigma$ is evaluated as true iff $\mathcal{A}(v) = \top$ (corresponding to equation 5)
- $E = c$ for some constant is evaluated as the constant (corresponding to equation 4)

1.2.2 CMOS Transistors

As the name would indicate, CMOS is made up of two distinct, complementary parts[3]: a P-transistor can only carry high potential from their source to their drain when the gate (connected to some wire representing a variable) is not driven, whereas an N-transistor can only carry low potential from their drain to their source when the gate is driven. Two power rails then provide a low potential (commonly referred to as V_{dd}, and in this document, referred to at an implementation level as `Drain`), and a high potential (commonly referred to as V_{ss}, and in this document, referred to at an implementation level as `Source`), with a third wire used as the output wire.

$$P(g, s, d) = \neg g \rightarrow (s \leftrightarrow d) \tag{8}$$
$$N(g, s, d) = g \rightarrow (s \leftrightarrow d) \tag{9}$$

Equation 8 describes a simplified version of the P-transistor: the potential is only carried (s matches d) when the input g is not powered, losing the notion that only high potential flows from the source to the drain. Equation 9 similarly describes a simplified version of the N-transistor: the potential is only carried (s matches d) when the input g is powered, losing the notion that only low potential flows from the drain to the source.

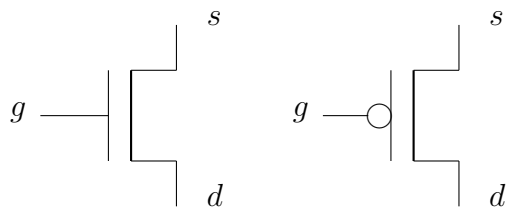


Figure 1: N-transistor and P-transistor, respectively

Therefore, when using these transistors, it is not sufficient to specify the implementation of a logical expression in a positive sense (carrying potential when the valuations of the inputs causes the expression to be evaluated as true), but rather, both the high potential must be driven to the output, and the low potential, using a complementary set of networks, made up solely of P-transistors and N-transistors. Throughout this report, and in the implementation, these are drawn with the high potential being carried from the top of a diagram to the middle by a network of P-transistors, and the low potential being carried from the bottom of a diagram to the middle by a network of N-transistors. To model conjunctions, a pair of P-transistors should be placed in series, indicating that the second transistor will only carry the potential if its gate is not driven, and the first transistor is carrying a high potential. Similarly, a disjunction is modelled by a pair of P-transistors placed in parallel: potential will be carried if at least one of the transistors' gate is not driven.

In a more general sense, for some expression e, we want a circuit $c = (p, n)$ such that $p \rightarrow e$ and $n \leftarrow e \equiv \neg n \rightarrow \neg e$, where p and n are entirely made of P- and N- transistors respectively.

Suppose that $a \wedge b$ is part of a network of P-transistors, such that p is the drain immediately above the implementation of $a \wedge b$, and q is the source immediately below the implementation. Therefore, p is acting as the source for $a \wedge b$, and q is the drain, and we therefore want a series of transistors such that $a \wedge b \rightarrow (p \leftrightarrow q)$. As described above, we will need the two transistors in series to give the necessary result, and they must be connected somehow, so

we therefore introduce a new variable to act as the point between them:

$$\begin{aligned} & a \wedge b \rightarrow (p \leftrightarrow q) \\ = \; & \exists x \cdot ((a \rightarrow (p \leftrightarrow x)) \wedge (b \rightarrow (x \leftrightarrow q))) \\ = \; & \exists x \cdot P(\neg a, p, x) \wedge P(\neg b, x, q) \end{aligned}$$

To define a similar construction in N-transistors, we have

$$\begin{aligned} & a \wedge b \leftarrow p \leftrightarrow q \\ \equiv \; & \neg(a \wedge b) \rightarrow (p \leftrightarrow q) \\ \equiv \; & \neg a \vee \neg b \rightarrow (p \leftrightarrow q) \end{aligned}$$

and we therefore derive

$$\begin{aligned} & \neg a \vee \neg b \rightarrow (p \leftrightarrow q) \\ = \; & (\neg a \rightarrow (p \leftrightarrow q)) \wedge (\neg b \rightarrow (p \leftrightarrow q)) \\ = \; & N(\neg a, p, q) \wedge N(\neg b, p, q) \end{aligned}$$

Alternatively, suppose that $a \vee b$ is part of a network of P-transistors, with p and q behaving as before, and we therefore want a series of transistors such that $a \vee b \rightarrow (p \leftrightarrow q)$. As described above, we will need the two transistors in parallel to give the necessary result:

$$\begin{aligned} & a \vee b \rightarrow (p \leftrightarrow q) \\ = \; & (a \rightarrow (p \leftrightarrow q)) \wedge (b \rightarrow (p \leftrightarrow q)) \\ = \; & P(\neg a, p, q) \wedge P(\neg b, p, q) \end{aligned}$$

To define a similar construction in N-transistors, we have

$$\begin{aligned} & a \vee b \leftarrow p \leftrightarrow q \\ \equiv \; & \neg(a \vee b) \rightarrow (p \leftrightarrow q) \\ \equiv \; & \neg a \wedge \neg b \rightarrow (p \leftrightarrow q) \end{aligned}$$

and we therefore derive

$$\begin{aligned} & \neg a \wedge \neg b \rightarrow (p \leftrightarrow q) \\ = \; & \exists x \cdot (\neg a \rightarrow (p \leftrightarrow x)) \wedge (\neg b \rightarrow (x \leftrightarrow q)) \\ = \; & N(\neg a, p, x) \wedge N(\neg b, p, x) \end{aligned}$$

The networks seen in fig. 2 on the following page are standard implementations of nand- and nor-gates.

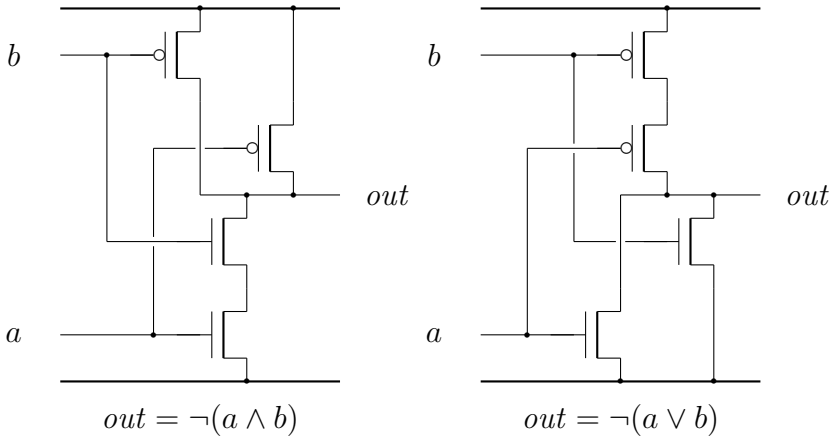


Figure 2: Standard nor- and nand- gate implementations

2 Requirements and Design

2.1 Requirements

The program would be expected to satisfy the following minimal requirements in order for it to be considered successful:

Accuracy any representation of CMOS produced must correctly implement the logical expression that has been entered, and the visual output of the expression should match the model that has been stored.

Representation the produced visualisation should be intuitive to read: each element should be presented separately from the others, and clearly labelled. Any gate carrying potential should be clearly highlighted.

Portability the visual output produced by the program should be available to the user in a variety of formats.

Interactivity the program should produce sensible warnings when an expression that is not possible to parse has been entered. Once a CMOS representation has been constructed, it should be possible for the user to interact with the publicly visible inputs (the variables named in the initial expression), with updates delivered with a minimal amount of latency. Should the user wish to view the change in state due to the latency of the transistors, this should be visible, and at a variable speed.

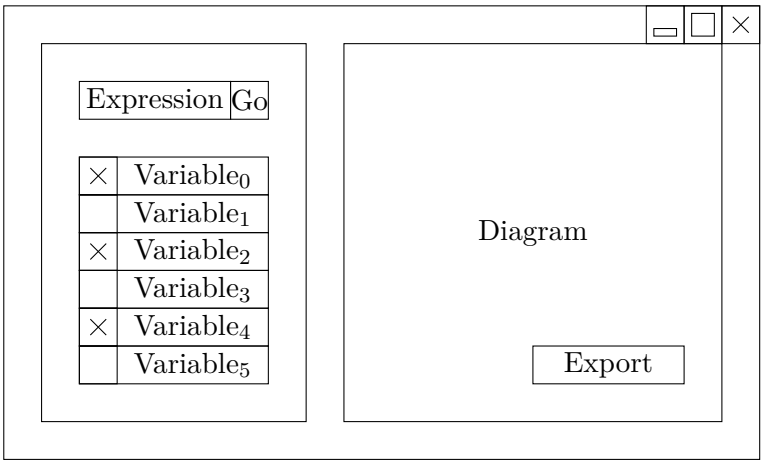


Figure 3: Initial Design

Given the nature of the project, the model-view-controller paradigm allows a sensible division of labour, also allowing for extensibility and clear separation of the purpose of the model and the interface.

2.2 Design

In order to fulfil the above specifications, it was obvious that a graphical interface would be required, so that the user could quickly interact with the program and determine the variable assignments intuitively, and to have their actions quickly reflected in the circuit diagram. Initially, due to the quality of the output produced by LaTeX diagrams, using the Tikz engine to produce the output and importing temporary files was considered, given the success of projects such as TikzEdt[10], however, as it would not be possible to assume that any system would have the necessary libraries and programs installed, it was determined that this would not be a feasible method to consider. Additionally, by using a Java-based drawing library, animations would be easier, should the project be extended to include them, as opposed to a LaTeX based solution, which is, by definition, designed for static documents.

2.3 Typical Use

On starting the application, the user will typically enter a well-formed logical expression, as described using the grammar above, with a non-trivial canonical representation (i.e. $expr \neq \top, \bot$). This will then be parsed and converted to a canonical form (DNF) which will then be processed to produce a series of transistors to implement the desired logical expression. This will then be rendered, with gates highlighted red to indicate drivenness, and a list of variables provided. The user can then change the assignment of these variables by toggling a checkbox, and this change will then be reflected in the rendered set of gates.

3 Implementation

3.1 Model

3.1.1 Logical Expressions

Every object that is used to represent logical expressions inherits from some common `Node` object that requires accessors to determine if the given fragment evaluates to true, as defined in Logical Expressions onwards. As would be expected given the recursive nature of the design of the grammar of the language of propositional logic, the parameters taken by each class that is not an atom are nodes, and these are then evaluated recursively.

Expressions are then parsed using Scala's Packrat parsing library. During parsing, each new variable is registered with a static map stored in the `Variable` object, that takes the place of the assignment function $\mathcal{A}$, as discussed in Logical Expressions. After parsing, but before the gate implementation (see Transistors and Wires), a modified version of the Quine-McCluskey algorithm (see Normal Forms) is applied to the parsed expression to minimise it and produce a canonical form.

The running time of the Packrat parser is consistently linear in the size of the input received, albeit at the cost of increased storage capacity, regardless of the specific grammar used.

3.1.2 Transistors and Wires

The transistors are similarly structured: inheriting common attributes, where possible, through a central `Transistor` class, sub-typing as appropriate.

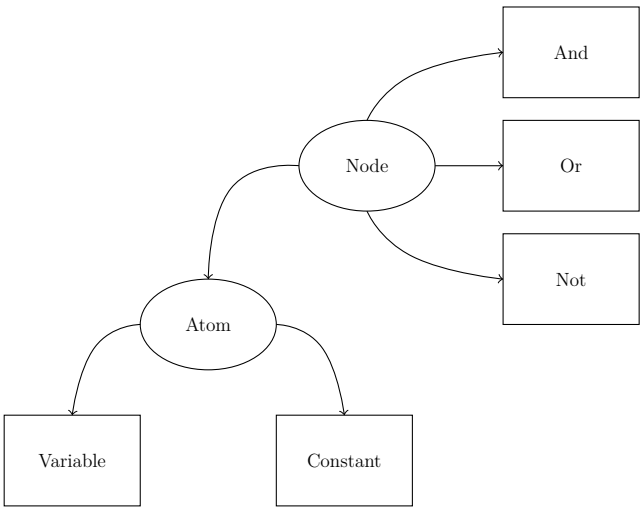


Figure 4: Logical Expression Class Relations

There are five different types of wire, all inheriting from a central `Wire` class. `Source` and `Drain` objects are responsible solely for delivering high and low potentials respectively, and a `Result` object acting as the central link. Finally, there are two wire classes responsible for carrying high and low potential downwards and upwards called `WireHigh` and `WireLow`, respectively.

The pseudo-code in algorithm 1 on the next page describes the final implementation of the gate creation algorithm, however, initially, a different approach was attempted, which expanded the number of classes inheriting from `Node` to include implications and a representation of iff, as well as the classes for the transistors. After parsing the formula, a function then proceeded to try and convert the formula and its negation into a series of transistors with literals as inputs, joined by conjunctions and disjunctions. Dealing with the large number of special cases quickly became impractical, especially when small repeated changes had to be made, and therefore was abandoned in favour of a separation of the logical and "physical" models, where the transistors were joined by wires and received potentials as inputs.

Once an initial array of transistors has been generated, the program then minimises the number of transistors produced, by traversing from `Source` to `Result`, followed by from `Drain` to `Result`, looking for transistors operating on the same literal that may be combined into one group, by removing the redundant transistor, and extending the wire below, as described in algorithm 2

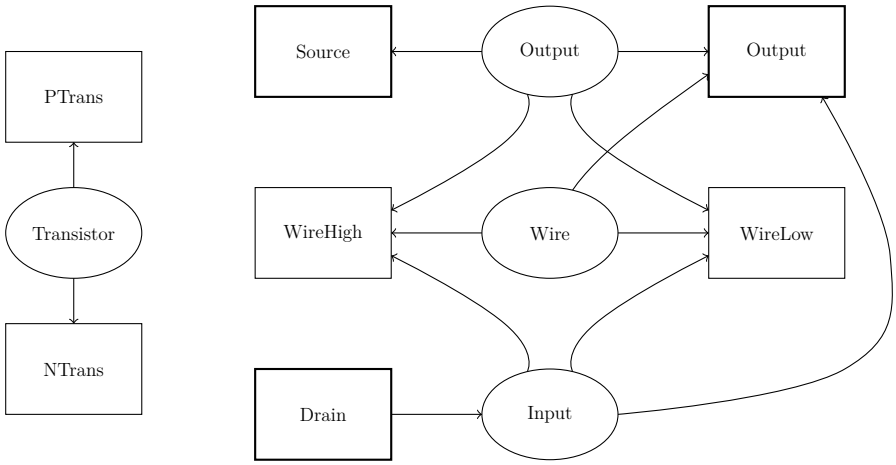


Figure 5: Gate and Wire Class Relations

Algorithm 1: Converting DNF expression to transistors

Input : Expression $E = \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m} P_i \right)$ in Disjunctive Normal Form

Output : None

Side Effects : `Result` has a set of transistors connected that implement E

clear Result;
previous-gate $\leftarrow$ ();
foreach $c_i \in E = c_1 \vee c_2 \vee \ldots \vee c_n$ **do**
    foreach $l_i \in c_i = l_1 \wedge l_2 \wedge \ldots \wedge l_m$ **do**
        if previous-gate *is defined* **then**
            Create a new gate and connect it to previous-gate, updating previous-gate to the new gate;
        else
            Create a new gate and connect it to Result, setting previous-gate to this gate;
    Connect previous-gate to Source/Drain as appropriate;

on page 14. Alternatively, rather than working inwards from the outer nodes, it could work outwards from the `Result` wire, however, it is likely that only one of these passes may be performed, as applying it in both directions may result in transistors being connected that should not be, e.g. $a \oplus b \oplus c$ will result in a and $\neg a$ being connected to the `Souce` twice, and c and $\neg c$ bein connected to the drain twice, however, by merging both sets of gates, connections are then formed such that `Result` is always driven.

Algorithm 2: Minimising the number of transistors

Input : Starting wire, top/bottom network indicator
Output : None
Side Effects : Minimised number of transistors in network, updated totals to reflect this

```
if wire ≠ Result then
    gates : V → Transistor ← ∅;
    if checkingTopNetwork then
        gate-list ← wire.getDrains
    else
        gate-list ← wire.getSources
    foreach gate ∈ gate-list do
        if gate.input ∈ dom(gates) then
            if checkingTopNetwork then
                foreach drain ∈ gate.drain.getDrains do
                    gates(gate.input).drain.addDrain(drain)
            else
                foreach source ∈ gate.source.getSources do
                    gates(gate.input).source.addSource(source)
            remove gate;
            total-gates ← total-gates − 1;
        else
            gates ← gates ∪ {(gate.input, gate)};
```

Additionally, it may be more expedient to negate the expression that needs finding and then negate the output to determine a final drivenness result, as if several input variables need negating, this will require an extra two

transistors per variable. For example, $a \wedge b$ requires 8 gates ($\neg a$ from `Source` to intermediate wire, $\neg b$ from intermediate wire to `Result`, $\neg a$ from `Drain` to `Result`, $\neg b$ from `Drain` to `Result`, and a pair of transistors to produce each of $\neg a$, and $\neg b$), whereas $\neg(\neg a \vee \neg b)$ requires 6 transistors (a from `Drain` to intermediate wire, b from intermediate wire to `Result`, a from `Source` to `Result`, b from `Source` to `Result`, and a pair of transistors to produce the final negation). It is therefore necessary to generate the transistor networks at least twice, however, it is far more useful to the program's structure to implement `Result` as a single static object, and in half of the cases, the worst-case of three passes will not be needed. This is discussed in more detail in algorithm 3.

Algorithm 3: Determining the optimal sequence of negations

Input : Expression to convert to transistors `expr`
Output : Integer value of number of gates needed
Side Effects : Transistor network produced on `Result`

```
normal ← try-layout(expr);
double-negate ← try-layout(¬expr) + 2;
if normal ≤ doubleNegate then
    try-layout(expr);
    return normal;
else
    return double-negate;
```

Initially, potential was going to be simulated using the `Option[Boolean]` type, with `Some(true)` and `Some(false)` to indicate high and low potential separately, and `Nothing` to simulate a transistor or wire that is not driven. This approach has also been reworked, to further separate the differences between the abstract logical expressions with precise true/false values, and the CMOS-specific high and low potentials where the transistors/wires are driven, or a specific `Undriven` value. Functionally, this does not differ significantly in practise from the originally proposed `Option[Boolean]` implementation. The pseudo-code in algorithm 4 on the following page, and algorithm 5 on page 17 describe in further detail the steps required to determine drivenness.

Algorithm 4: Traversing transistors to determine drivenness

Input : None (necessary information contained in static object)
Output : True/false value to indicate if the output is driven
Side Effects : None

```
r ← Undriven;
foreach transistor ∈ Result sources do
    if transistor-Status(gate) ≠ Undriven then
        r ← transistor-Status(transistor);
if r = Undriven then
    foreach transistor ∈ Result drains do
        if transistor-Status(gate) ≠ Undriven then
            r ← transistor-Status(transistor);
if r = Undriven then
    report error
else
    return r
```

3.1.3 Helpers

Parsing The Packrat parsing library is one of the many libraries available in Scala for parsing purposes.[4] This library has the advantage of allowing efficient parsing by using memoization to reduce the number of times that a production rule needs testing. This parsing library uses a domain-specific language to assist in the definition of the grammar to be parsed to ensure that the abstract representation of the grammar matches the implementation as closely as possible.

Normal Forms After an expression is parsed, it is then converted to a normal form to allow the transistor conversion tool to process it, in this case, as a disjunction of conjunctions, by using the Quine-McCluskey Algorithm, and algorithm developed by McCluskey[6], building on work by Quine[7, 8]. The algorithm has the following sections:

1. Convert logical expression to boolean function:

$$\begin{aligned} f &: \{0,1\}^{|\mathcal{A}|} \to \{0,1\} \\ f(v_1, v_2, \ldots v_n) &= \sum m(n_1, n_2, \ldots, n_m) + d(n_{m+1}, \ldots, n_k) \end{aligned}$$

Algorithm 5: Helper methods to determine drivenness

```
def transistor-Status (transistor: Transistor) as
    Input  : transistor to check for drivenness
    Output : Potential on transistor
    r ← Undriven;
    if transistor is P-transistor then
        if ¬transistor.input then
            r ←Wire-Status(transistor.source)
        else
            r ← Undriven
    else
        By symmetry for N-transistor
    return r;
def Wire-Status (wire: Wire) as
    Input  : wire to check for driven-ness
    Output : Potential on wire
    r ← Undriven;
    if wire is Source then
        r ← High
    else if wire is Drain then
        r ← Low
    else if wire is WireHigh then
        if ¬transistor.input then
            r ←Wire-Status(transistor.source)
        else
            r ← Undriven
    else if wire is WireLow then
        By symmetry for WireLow
    return r;
```

where m is the set of minterms, and d is the set of "don't care" values (not needed in this implementation, as we are only considering logical expressions that are defined in the syntax given above). The minterms give a non-minimal canonical representation of a formula (e.g. for $f(A, B) = \sum m(0)$, f represents the boolean formula $\neg A \wedge \neg B$). This is described in more detail in algorithm 6 on the following page.

2. Find the prime implicants of the function (P is an *implicant* for F if P is a conjunction of literals, and if $\mathcal{A} \models F$ whenever$\mathcal{A} \models P$. A *prime implicant* is an implicant that is minimal in the number of terms that it contains (i.e. no more literals can be removed without it becoming a non-implicant)). This is described in more detail in algorithm 7 on page 20.

3. Find the essential prime implicants, as well as any others that are necessary to cover the function (a prime implicant is *essential* when it is the only prime implicant to cover some minterm). This is described in more detail in algorithm 8 on page 21.

4. Print/export the results using the list of variables to assign values, and iterating through each of the essential implicants. This will also result in the returned expression's clauses already being in alphabetical order, making later optimisations based on connections of gates easier.

When processing a logical expression by hand, the usual method is to produce a Karnaugh map, first discussed in [5], and use manual techniques to quickly find minterms that cover as much as possible. For example, the Karnaugh maps for $(a \wedge b) \vee (\neg a \wedge \neg b \wedge c)$, and $a \oplus b \oplus c$ are produced in table 1 on page 21. While the first expression, and its negation, decomposes nicely into a series of clauses where each clause covers at least one other box, and usually two, $a \oplus b \oplus c$ is an example of a degenerate case without any such nice decomposition, but rather, each clause produced covers one box at a time.

3.2 View and Controller

The view and controller are handled by two separate classes, one with an overall responsibility for drawing and updating the UI view presented to the user, written in Java and named `Gui`, which then invokes the `DrawCircuit`

Algorithm 6: Quine-McCluskey – Logical Expression to Boolean Function

Input : Logical Expression
Output : List of minterms

Get from $\mathcal{V}$ from Expression;
Variable list $\leftarrow$ "";
foreach $v \leftarrow \mathcal{V}$ **do**
  Variable list $\leftarrow v +$ Variable list;
foreach $i \in \{0, 1, \ldots, 2^{|\mathcal{V}|}\}$ **do**
  foreach $j \in \mathbb{Z}_{|\mathcal{V}|}$ **do**
    $\mathcal{A}(v_j) = {}^{i}/_{2^j} \mod 2$;
  if $\mathcal{A} \models$ Expression **then**
    Minterms $\leftarrow$ Minterms $\cup \{i\}$
return Minterms

Scala class to initially draw and then update the displayed circuit as the inputs are changed. The following algorithm is a naïve version, designed for drawing transistors where each wire, other than the static `Source`, `Drain`, and `Result` has precisely one transistor attached. This is described in more detail in algorithm 9 on page 22.

In order to modify the above algorithm to draw transistors where one wire may have multiple transistors attached, the drawing algorithm checks for the presence of a pre-drawn transistor, and if it is found, connects the two branches and finishes drawing (as it would otherwise duplicate work). This does increase the work necessary before drawing, however, as the presence of drawn transistors needs to be removed, using a visitation pattern not dissimilar to the method used to determine drivenness. This is described in more detail in algorithm 10 on page 23.

Unfortunately, this algorithm does not properly take into account overlapping transistors/wires, as they are drawn from the `Result` wire outwards. Techniques to allow both the optimised drawing algorithm, and the gate-reduction algorithms to work together more effectively are discussed in section 6 on page 30.

Algorithm 7: Quine-McCluskey – Prime Implicants

Input : List of minterms minterms

Output : List of prime implicants

foreach $i \in$ minterms **do**
  trueCount $= 0$;
  foreach $j \in \mathbb{Z}_{|\mathcal{V}|}$ **do**
    implicant-table$_{i,v_j} \leftarrow {}^{i}/_{2^j} \mod 2$;
    trueCount $\leftarrow$ trueCount $+ {}^{i}/_{2^j} \mod 2$;
  implicant-table$_{i,\text{trueCount}} \leftarrow$ trueCount;
modified $\leftarrow \top$;
while *modified* $= \top$ **do**
  modified $\leftarrow \bot$;
  foreach $i \in$ implicant-table **do**
    foreach $j \in$ implicant-table, $j \neq i$ **do**
      if *i and j differ in one position in* implicant-table **then**
        implicant-table$_{\langle i,j \rangle} \leftarrow$ implicant-table$_i \oplus$ implicant-table$_j$
        where $i \oplus j$ produces $i \wedge j$ with any differences replaced
        with –;
        modified $\leftarrow \top$;
    if *modified* $= \bot$ **then**
      mark i as prime;
return marked entries in implicant-table

Algorithm 8: Quine-McCluskey – Essential and covering implicants

Input : List of prime implicants, and minterms

Output : Minimised expression in DNF

foreach $m \in$ Minterms **do**
  foreach $p \in$ prime-implicants **do**
    if $m \models p$ **then**
      minterm-imp-table$_{m,p} \leftarrow \top$;

covered-minterms $\leftarrow \emptyset$;

chosen-implicants $\leftarrow \emptyset$;

foreach $m \in$ Minterms **do**
  if $| x = \{i \in$ minterm-imp-table : minterm-imp-table$_{m,i} = \top\} | = 1$ **then**
    minimised-minterms $\leftarrow$ minimised-minterms $\cup\, x$;
    foreach *minterm* $\in x$ **do** Find all essential implicants, and add them to the list of required implicants
      covered-minterms $\leftarrow$ covered-minterms $\cup$ minterm;

foreach $m \in$ Minterms $\setminus$ covered-minterms **do**
  Find $c \in$ prime-implicants such that c covers m and as many other values as possible;
  minimised-minterms $\leftarrow$ minimised-minterms $\cup \{c\}$;

return minimised-minterms

		ab			
		00	01	11	10
c	0	0	0	1	0
	1	1	0	1	0

		ab			
		00	01	11	10
c	0	0	1	0	1
	1	1	0	1	0

		ab			
		00	01	11	10
c	0	0	0	1	0
	1	1	0	1	0

		ab			
		00	01	11	10
c	0	0	1	0	1
	1	1	0	1	0

Table 1: Karnaugh Maps

Algorithm 9: Transistor Drawing Algorithm

Input : Graph object to draw on (all other data accessed statically)

Output : None

Side Effects : Graph updated with visual representation of transistors

```
Clear Graph;
Add Result to graph as long thin horizontal line going through (0, 0);
x ← 0;
foreach transistor : N-transistor attached to Result do
    y ← 0;
    current-transistor ← transistor;
    while current-transistor ≠ Source do
        Add current-transistor to Graph at (x, y) with size (δ_x, δ_y);
        if transistor-Status(current-transistor) ≠ Undriven then
            Highlight current-transistor
        current-transistor ← transistor.next;
        y ← y + δ_y;
    last-transistors ← last-transistors ∪ current-transistor;
    x ← x + δ_x;
The above, by symmetry, for the bottom transistor network;
foreach transistor ∈ last-transistors do
    if transistor : N-transistor then
        Connect transistor to Source;
    else
        Connect transistor to Drain;
```

Algorithm 10: Optimised Transistor Drawing Algorithm

Input : Graph object to draw on (all other data accessed statically)

Output : None

Side Effects : Graph updated with visual representation of transistors

```
Clear Graph;
Traverse transistors to reset drawn-transistor property;
drawn-transistors ← ∅;
Add Result to graph as long thin horizontal line going through (0, 0);
x ← 0;
foreach transistor : N-transistor attached to Result do
    y ← 0;
    current-transistor ← transistor;
    while current-transistor ≠ Source do
        if current-transistor.drawn-node= ∅ then
            Add current-transistor to Graph at (x, y) with size (δx, δy);
            drawn-transistors ← drawn-transistors ∪ {current-transistor};
            if transistor-Status(current-transistor)≠ Undriven
            then
                Highlight current-transistor
            current-transistor ← transistor.next;
            y ← y + δy;
        else
            Connect previous-transistor to current-transistor.drawn-node;
            current-transistor← Source;
    last-transistors ← last-transistors ∪ current-transistor;
    x ← x + δx;
The above, by symmetry, for the bottom transistor network;
foreach transistor ∈ last-transistors do
    if transistor : N-transistor then
        Connect transistor to Source;
    else
        Connect transistor to Drain;
```

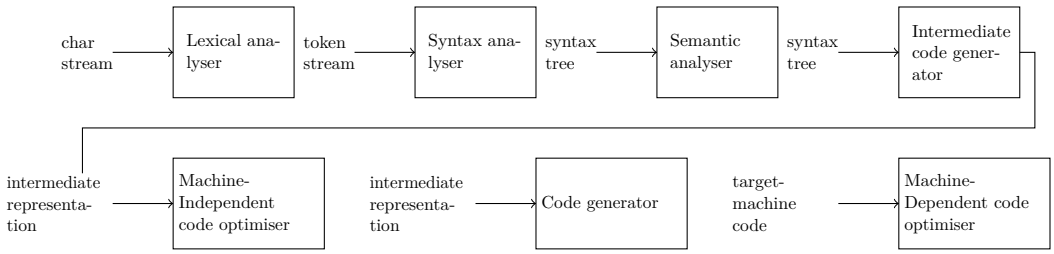


Figure 6: Compiler Structure

4 Comparison to a Compiler

The usual definition of a compiler is "a program that can read a program in one language and translate it into an equivalent program in another language" [1] usually with some sort of intermediate feedback if errors are encountered; with most compilers usually translating from a human-readable, high-level language into something low-level that can then be directly executed by a machine. The structure of a compiler can be thought of as the composition of a series of functions, each with a distinct focus, as shown in Figure 6 on page 24.

In a conventional compiler, the lexical analysis stage splits up the entered text into variables to be recognised within the program, or as keywords/reserved tokens within the language, usually achieved using some deterministic finite automata to recognise a regular expression, with some order of precedence established to allow keywords to be treated as such, rather than as variables. This is followed by syntax analysis, where the discrete tokens are then used to form a *syntax tree*, where the nodes are production rules or terminals in the context-free grammar, and the edges correspond to variables leading to further production rules, similar to the structure of logical expressions defined in (1) to (5) in eq. (4) on page 5. Semantic analysis varies in its depth, depending on the language, however, most compilers will usually perform some sort of check for proper declaration, as well as ensuring that variables are not used outside of the scope that they were defined in.

The lexical, syntax, and semantic analysis stages can all be thought to be covered by the parsing stages in the application, as the various tokens in the language (`and`, `or`, `!`, `(`, and `)`) are isolated and then used to form the syntax tree, while variables are added to the assignment map in the `Variable` object. Additionally, should any stage of the parsing module fail, the first of

these errors is then reported back to the user to allow them to correct their input, and the assignment map emptied so that their previous state does not then affect their next input. Owing to the simplicity of the propositional logic language, there is obviously no need for formal semantic analysis of the entered expression, however, it is useful to register each variable, with the minimal semantic check that the reserved keywords `out`, `and`, and `or` are not used in the entered expression.

Once no more work can occur on the syntax tree, the compiler then starts to focus on generating code, directed by the structure of the syntax tree, following by platform-agnostic optimisations that focus on repeated patterns within the code, or redundant operations often focusing on a small number of operations at any one time. This process is often called *peep-hole optimisation* owing to the metaphor of only being able to see a small part of a scene at any one time through a peep-hole camera.

This is replaced in the CMOS calculator by the Quine-McCluskey algorithm before transistor generation, effectively putting the optimisation before the code generation, though in a more general software engineering metaphor, this could be considered as a refactoring to remove redundancy before actually compiling the code, resulting in different byte-code, but with equivalent effects. Additionally, this has the advantage of ensuring that expressions such as $\bigwedge_{i=0}^{n} a$ get reduced properly to a, rather than attempting to produce n N-transistors in series for , followed by n P-transistors in parallel, so the number of gates is strictly bound by the number of variables, rather than the size of the entered expression.

Once no more work can take place on the intermediate code, a compiler then starts producing platform-specific code, followed by any further platform-specific operations. In the CMOS calculator's case, this would then be the transistor generation algorithm, taking the canonical form generated by the Quine-McCluskey algorithm, and turning it into the CMOS implementation.

The next stage in the pipeline produces optimisations to the machine-readable code that has been produced for the relevant platform, with a specific focus on architecture specific optimisations. For Intel-based systems, with their larger instruction sets, and instructions capable of processing multiple operations on data at a time, this may focus on maximising the number of pieces of data operated upon by combining them into vectors, or on ARM machines with large numbers of registers, this frequently translates into taking advantage of the large caches available to minimise the number of secondary memory lookups and therefore the number of cycles spent idling while the

RAM is queried. Within the context of the CMOS transistors, this then focuses on maximising the number of transistors that may be eliminated while retaining the same original expression. Just as the optimisation stages of a compiler will focus on the structure of the program to find subtle optimisations, so too does this program produce a minimised final output by examining the wires nearest to it.

5 Testing

5.1 General Testing Strategy

The output from the logical expression classes was deliberately designed so that the output that they would produce would be compatible with the parser, thus making testing the output of any given module producing logical expressions as output much easier, as they could then be compared to the expected values, once the behaviour of the parser was verified.

Where possible the testing is designed to test error cases, simple cases (e.g. atoms in the parser, or simple expressions for the Quine-McCluskey algorithm), followed by more complicated cases, usually based on the inductive definitions seen above.

5.2 Parsing

Care must be taken, as already discussed to ensure that only valid inputs are parsed, without affecting the ability of the program to operate using predefined constants that have been set aside for specific purposes. Furthermore, the parser must ensure that statements are parsed with due care paid to the order of operations, with negations binding correctly. The contents of table 2 on the following page list the sample test data and expected outcomes.

5.3 Quine-McCluskey Algorithm

The contents of table 3 on page 28 list the sample test data and expected outcomes.

Input	Expected Output	Input	Expected Output
“”	None	a and b	And(Variable(a), Variable(b))
0	False	!a and b	Not(And(Variable(a), Variable(b))
1	True	a and !b	And(Variable(a), Not(Variable(b)))
out	None	!a and !b	Not(And(Variable(a), Not(Variable(b)))
test	Variable(test)	(!a) and b	And(Not(Variable(a)), Variable(b))
(	None	a and (!b)	And(Variable(a), Not(Variable(b)))
)	None	(!a) and !b	And(Not(Variable(a)), Not(Variable(b)))
()	None	or	None
a	Variable(a)	a or	None
(a)	Variable(a)	or a	None
!	None	a or b	Or(Variable(a), Variable(b))
!a	Not(Variable(a))	!a or b	Not(Or(Variable(a), Variable(b)))
(!a)	Not(Variable(a))	a or !b	Or(Variable(a), Not(Variable(b)))
!(a)	Not(Variable(a))	!a or !b	Not(Or(Variable(a), Not(Variable(b))))
and	None	(!a) or b	Or(Not(Variable(a)), Variable(b))
a and	None	a or (!b)	Or(Variable(a), Not(Variable(b)))
and a	None	(!a) or !b	Or(Not(Variable(a)), Not(Variable(b))))

Table 2: Parsing Test Data

Input	Expected Output
(!a) and a	None
(!a) or a	None
a and a	a
a or a	a
((!a) and b) or (a and (!b))	Or(And(Variable(a), Not(Variable(b))), And(Not(Variable(a)), Varia

Table 3: Parsing Test Data

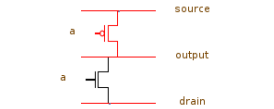
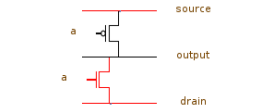
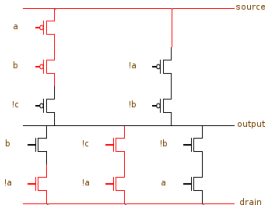
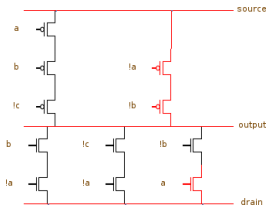
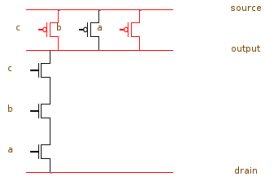
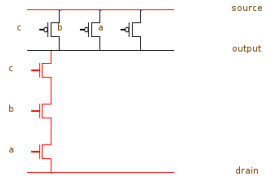
Input	Driven Output	Undriven Output	Transistors
$\neg a$			2
$(a \wedge b) \vee (\neg a \wedge \neg b \wedge c)$			17
$a \wedge b \wedge c$			$6 + 2$

Table 4: Program Output

5.4 Transistor Generation and Drawing

Due to the subjective nature of a "good" layout, the diagrams in table 4 demonstrate the result of entering the following expressions. When the number of transistors needed is formatted as $n + 2$, this is used to indicate that less transistors are needed to produce the negation of the expression and negate this, rather than to produce the original expression. For clarity, in the case of a tie, the original expression is preferred.

6 Conclusions and Further Work

6.1 Conclusions

The project has achieved much of the scope that it had originally set out to, however, the order of completion of different subcomponents of the program needed changing quite substantially. Initially, the order of completion of the modules was intended to be parser, transistor-generator, gate-drawer, optimisations (including Quine-McCluskey-related functions), however, it quickly became apparent that laying out the transistors would be far quicker and simpler if the input was in some sort of normal form first, therefore, finding an implementation of the Quine-McCluskey algorithm became a far higher priority, with drawing and a finalised transistor-generation algorithm being slightly delayed. On the other hand, as predicted, once a gate-drawing algorithm was in place, it became far easier to troubleshoot issues relating to local optimisation of gates, as the output of the optimisation sequence was far easier to visualise, rather than attempting to traverse the representation of wires and transistors stored in the currently executing program's memory.

Accuracy The program's visual output, as the drawing algorithm is structurally based on the representation within the model, the rendering is accurate. Owing to constraints on the drawing algorithm, it was necessary to sacrifice an optimal number of gates in lieu of ensuring that the drawn output remains consistent and easily interpreted: due to the way that mxGraph positions vertices and edges, any attempts to reduce overlap would have caused the edges to shift and become disconnected.

Representation The program produces a clearly labelled network of transistors, with driven transistors indicated in red, and undriven transistors in black. Unlike the diagrams seen in fig. 2 on page 9, wires linking common variables are not shown, to further enhance clarity, as in more complicated networks, a large number of overlapping wires may be needed. Additionally, when a variable needs negating, the small not-gate is not shown, nor are the transistors necessary, as this would only complicate the number of diagrams needed, as well as their layout.

Portability Beyond the bitmap based images that are produced by the program, which are then easily converted between similar formats, there is no other output produced by the program, to either allow reloading

old networks, nor raster-based images to allow simple scaling without loss of resolution. While a number of workarounds exist to solve the latter problem, through software that is already in existence, these are likely to produce images that are less than optimally encoded, as they will not necessarily take into account. Additionally, unlike 9-patch image files, stretching in one dimension will produce a skewed image, rather than sub-components resizing themselves intelligently.

Interactivity The program produces several different types of warning, corresponding to each stage of the compilation, and then the current runtime state of the circuit. The earliest warning may be invoked during parsing if a incorrectly parsed expression is seen, followed by after the completion of the first stage of the Quine-McCluskey algorithm, if the entered logical expression has a trivial normal form ($\top$ or $\bot$). Finally, in the event that the circuit produced does not always drive the output with a high or low potential, this will cause a warning to be thrown as it occurs, however, this may be a transient error, as the circuit should only be driven high or low at any time. If in the future, it becomes possible to import pre-produced circuits, some of these may not be properly formed, so it is a helpful check to have in place, as well as when prototyping the transistor production algorithm to ensure that the output was driven consistently.

6.2 Further Work

6.2.1 Model

This project has deliberately focused on defining logical expressions using strictly binary values, and precisely adhering to the standard rules for laying out CMOS diagrams, without any deviations. It has been assumed that the output wire will always be driven, with an undriven state being invalid, whereas the standard tri-state buffer [3]gate, where $enb \rightarrow (in \leftrightarrow out)$, and $\neg enb$ results in the output wire being undriven.

Additionally, the structure of the transistors produced by the transistor placing algorithm Transistors and Wires is more constrained than the transistors that may be drawn.

Decreasing the verbosity of the required input should also be a priority for future development. While operators such as $\rightarrow$, $\leftrightarrow$, $\oplus$ do not add any

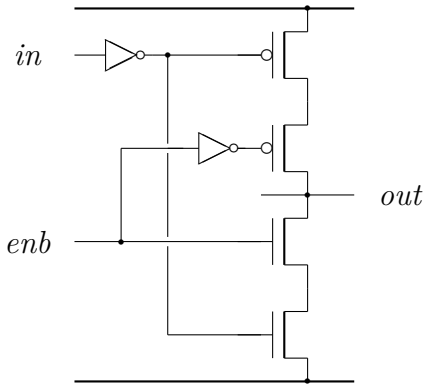


Figure 7: Tri-state buffer

additional expressiveness to the language, they are not currently parsed by the parsing module.

Furthermore, exploring limits on the size of the circuits produced to more accurately simulate the effects of resistance could be a further expansion for the system, which would also necessitate changes to the procedure by which the visual output is created in order to facilitate animation to properly demonstrate the effect of resistance in the time taken for the current to flow across a transistor or through wires, assuming non-negligible resistance. A naïve approach would involve the height of series of transistors being artificially limited, however, this would not take into account the length of connecting wires.

The model could also be expanded to include a more formalised notion of Tony Hoare's drivenness analysis[2], as discussed in a draft paper, where formulae (8) and (9) in section 1.2.2 on page 6 are updated as follows:

$$\delta P(g, s, d) = \neg g \wedge s \wedge d \wedge \delta g \rightarrow (\delta s \leftrightarrow \delta d) \quad (10)$$

$$\delta N(g, s, d) = g \wedge \neg s \wedge \neg d \wedge \delta g \rightarrow (\delta s \leftrightarrow \delta d) \quad (11)$$

where the presence of δv is used to indicate that v is driven, either high or low, if δv also holds.

6.2.2 View

The data produced by the view is currently only in a limited number of bitmap image formats, whereas a future version of this software should be able to produce a wider variety of formats, and ideally, especially for reports like this, LaTeX-compatible output, or Dia compatible output.

A future implementation of the transistor-drawing algorithm should be reworked in order to draw the transistors from the `Source` and `Drain` objects respectively, so as to ensure that grouped transistors are handled properly, without overlapping, however, this would also require a more detailed knowledge as to the structure and bounds of the generated network of transistors.

6.2.3 Controller

Due to the implementation of the pointer-based mxGraph that I was using to produce the visual output, it was not possible for the user to interact with the graph itself directly, rather, and changes to the assignments or expression triggered a redraw where any changes were then made visible. In larger graphs, if they were particularly complicated, this may not have always been particularly obvious, therefore, a future iteration of this project, a brief highlighting effect should be considered, in order to further improve visibility. Additionally, due to the indirect way of accessing nodes in the graph, a useful addition would be to include the ability to directly interact with the graph to see the result of changing specific variables.

Adding the ability to drag-and-drop transistors onto the graph, connect them and derive an expression for a valid arrangement of transistors would be a logical extension of the current functionality, though it is likely that any output produced would, unless the number of variables was limited, not be in any normal form, due to the cost of running the Quine-McCluskey algorithm (in the worst case, $(\mathcal{O}\left({}^{3^n}/_{n}\right))$ prime implicants will be generated alone). Additionally, unless the graph was encoded with the rules required for valid CMOS structures, it would be trivial for users to produce graphs that may have undriven output, which then cannot be validly encoded as a logical expression.

7 Acknowledgements

There are a number of people, without whose assistance over the past year, this project would not have been possible.

Professor Geraint Jones, my supervisor, for his advice and support, as well as the diagrams of the N-transistors and P-transistors seen in this document.

Matthew Sjödin, Eleanor Kirk, and Stephen Heap for general support throughout the project.

References

[1] Alfred V Aho. *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. Pearson Education India, 2003. 24

[2] C. A. R. Hoare. A calculus for the derivation of C-MOS switching circuits. Two drafts; appeared as 'A Theory for the Derivation of Combinational C-MOS Circuit Designs', *Theoretical Computer Science* 90:235–251, 1991, doi 10.1016/0304-3975(91)90309-P, April – October 1988. 32

[3] Geraint Jones. Digital Systems, CMOS. Lecture notes, April 2013. Compiled by report author. 6, 31

[4] Manohar Jonnalagedda. Packrat parsing in scala. Online, January 2009. 16

[5] M. Karnaugh. The map method for synthesis of combinational logic circuits. *American Institute of Electrical Engineers, Part I: Communication and Electronics, Transactions of the*, 72(5):593–599, Nov 1953. 18

[6] Edward J. McCluskey. Minimization of boolean functions. *Bell System Technical Journal*, 35:28, November 1956. 16

[7] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):pp. 521–531, 1952. 16

[8] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):pp. 627–631, 1955. 16

[9] Bernard Sufrin. Introduction to Formal Proof. Lecture notes, April 2013. Compiled by report author. 5

[10] T Willwacher. Tikzedt – a semigraphical Tikz editor. Online. 10

A Appendix

Listings

A.1 Model

Listing 1: And.scala

```
0 package model

  /**
   * Created by joshua on 17/12/14.
   */
5 case class And(lhs : Node, rhs : Node) extends Node {
    def get = lhs.get && rhs.get
```

```
  // Allows for commutativity of operators
  override def equals(that : Any) : Boolean = that match {
    case And(lhs_, rhs_) => (lhs_ == lhs && rhs_ == rhs) ||
      (lhs_ == rhs && rhs_ == lhs)
    case _ => false
  }

  override def toString() = (lhs match {
    case Variable(x) => x;
    case _ => "(" + lhs.toString + ")";
  }) + " and " + (rhs match {
    case Variable(x) => x;
    case _ => "(" + rhs.toString + ")";
  })
}
```

Listing 2: Atom.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
trait Atom {
  def get() : Boolean
}
```

Listing 3: Constant.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
case class Constant(val truth : Boolean) extends Node with
    Atom {
  def get = truth
}
```

Listing 4: Drain.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
object Drain extends Wire with Input {
  override def removeGate (transistor : Transistor) : Unit
     = {
    sources = sources diff Array(transistor)
  }

  def get() : Potential = Low()

  override def clear : Unit = clearSources

  override def toString() = "Drain"

  override def resetDrawnGates () : Unit = ()
}
```

Listing 5: Driven.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
trait Driven {
}
```

Listing 6: High.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
case class High() extends Driven with Potential {
```

```
  override def isHigh : Boolean = true
}
```

Listing 7: Iff.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
case class Iff(lhs : Node, rhs : Node) extends Node {
  def get = (!lhs.get || rhs.get) && (!rhs.get || lhs.get)
}
```

Listing 8: Implicant.scala

```
package model

/**
 * Created by joshua on 23/12/14.
 */
class Implicant(val minterm : Int, val tag : Int = 1, val
     group : List[Int] = Nil) {
  var prime : Boolean = true

  def cost(order : Int) : Int = {
    order - group.size
  }

  def order() : Int = {
    return group.length
  }

  def canCombine(other : Implicant) : Boolean = {
    //--- if the other one is less than this, don't bother
     comparing
    //if (other.minterm < minterm)
    //return false

    //--- only include ones that exist in at least one
     function
    if ((other.tag & tag) == 0)
      return false

    //--- if differences are not equivalent, don't bother
     comparing
    if (group != other.group)
      return false

    def bitdist(x : Int, y : Int) = qmm.bitcount(x ^ y)

    //--- difference needs to be just one bit
    if (bitdist(other.minterm, minterm) != 1)
      return false

    return true
  }

  override def equals(that : Any) = that match {
    case other : Implicant => {
      hashCode == other.hashCode
    }
    case _ => false
  }

  override def hashCode = terms().hashCode

  def combine(other : Implicant) : Implicant = {
    val newtag = other.tag & tag;
    val diff = math.abs(other.minterm - minterm)
    val newgroup = (group ::: List(diff)).sorted
    val newmt = if (minterm > other.minterm) other.minterm
     else minterm

    return new Implicant(newmt, newtag, newgroup)
  }

  override def toString() = terms().mkString("<", ",", ">")

  def printTerms() = println(terms().mkString("(", ",", ")"
     ))
```

```
  def terms() = {
    var terms : List[Int] = List(minterm)
    for (difference <- group) {
      terms = terms ::: terms.map(_ + difference)
    }
    terms
  }

  def print() {
    printf("%d %s tag=%d %s\n", minterm, group.mkString("("
     , ",", ")"), tag, if (prime) {
      "prime"
    } else {
      ""
    })
  }

  def withVars(vars : List[String]) : String = {
    val weights = (0 until vars.length).map(1 << _).reverse
    val varByWeight = (weights zip vars).toMap
    //println(varByWeight)

    val expression = for (w <- weights) yield {
      if (!group.contains(w)) {
        if ((minterm & w) != 0) {
          varByWeight(w)
        } else {
          "!(%s)".format(varByWeight(w))
        }
      } else {
        ""
      }
    }

    expression.filter(_ != "").reduceLeft(_ + " and " + _)
  }
}
```

Listing 9: Impl.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
// These should not be created when parsing, but are only
     used by the cmosify function
case class Impl(lhs : Node, rhs : Node) extends Node {
  def get = !lhs.get || rhs.get
}
```

Listing 10: Input.scala

```
package model

/**
 *
 */
trait Input {
}
```

Listing 11: LogicalFunction.scala

```
package model

import helper.Parser

/**
 * Represents the current logical expression as a sum of
     products
 * (sum of minterms)
 */
object LogicalFunction {
  private[this] var minterms = List[List[Variable]]()
  private[this] var satisfying = Set[List[Variable]]()

  def get() : Boolean = {
    var result = false;
    for (term <- minterms) {
      var minres = true;
```

```
      for (atom <- term) {
        minres = minres && atom.get
      }
      result = result || minres
    }
    result
  }

  def quineMcCluskey(expr : Node) : Option[Node] = {
    Parser.variableParser(convertToMinTerms(expr))
  }

  def convertToMinTerms(expr : Node) : String = {
    var numericalValues = List[Int]()
    val identMap = Variable.getMap - "out"
    val size = 1 << identMap.size
    var variables = ""
    for (term <- identMap) {
      variables = term._1 + variables
    }
    // iterate over every combination of values (2^n time)
    for (i <- 0 until size) {
      var j = i
      var terms = List[Variable]()
      for (term <- identMap) {
        if (j % 2 == 1) {
          terms = Variable(term._1) +: terms
          Variable.setValue(term._1, true)
        } else {
          Variable.setValue(term._1, false)
        }
        j = j / 2
      }
      if (expr.get) {
        satisfying = satisfying + terms
        numericalValues = i +: numericalValues
      }
    }
    return qmm.method(numericalValues, Nil, qmm.letters(
     variables))
  }

  def main(args : Array[String]) {
    //println(convertToMinTerms(Parser.variableParser("(a
     and b) or (a and !b) or (!a and b)") match { case Some
     (x)
    // => x }))
    println(convertToMinTerms(Parser.variableParser("(a and
      !c) or (!a and b)") match { case Some(x) => x }))
  }
}
```

Listing 12: Low.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
case class Low() extends Driven with Potential {
  override def isHigh : Boolean = false
}
```

Listing 13: Node.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
abstract class Node {
  var isInNormalForm = false;

  def get() : Boolean
}
```

Listing 14: Not.scala

```
package model
```

```scala
/**
 * Created by joshua on 17/12/14.
 */
case class Not(negated : Node) extends Node with Atom {
  def get = !negated.get

  override def toString = "!" + (negated match {
    case Variable(_) => negated.toString
    case _ : Node => "(" + negated.toString + ")"
  })
}
```

Listing 15: NTrans.scala

```scala
package model

/**
 * N(g, s, d) = g -> (s <-> d)
 *
 * N gates can only carry low potential from their drain to
     their source
 */
case class NTrans(input : Node, drain : Wire, source : Wire
    ) extends Transistor {
  override def get : Potential = if (input.get) drain.get()
      else Undriven()
  var drawnGate : Option[AnyRef] = None

  def sourceDriven = input.get && (drain.get == Low())

  override def resetDrawnGates() : Unit = {
    drawnGate = None
    drain.resetDrawnGates()
  }
}
```

Listing 16: Or.scala

```scala
package model

/**
 * Created by joshua on 17/12/14.
 */
case class Or(lhs : Node, rhs : Node) extends Node {
  def get = lhs.get || rhs.get

  // Allows for commutativity of operators
  override def equals(that : Any) : Boolean = that match {
    case Or(lhs_, rhs_) => (lhs_ == lhs && rhs_ == rhs) ||
     (lhs_ == rhs && rhs_ == lhs)
    case _ => false
  }

  override def toString() = (lhs match {
    case Variable(x) => x;
    case _ => "(" + lhs.toString + ")";
  }) + " or " + (rhs match {
    case Variable(x) => x;
    case _ => "(" + rhs.toString + ")";
  })
}
```

Listing 17: Output.scala

```scala
package model

/**
 * Created by joshua on 17/12/14.
 */
trait Output {
}
```

Listing 18: Paren.scala

```scala
package model

/**
 * Created by joshua on 17/12/14.
 */
case class Paren(expr : Node) extends Node {
```

```
  def get = expr.get
}
```

Listing 19: ParserTest.scala

```
/**
 * Created by joshua on 31/12/14.
 */

package model.test

import helper._
import model._
import org.scalatest.prop.{GeneratorDrivenPropertyChecks,
    TableDrivenPropertyChecks}
import org.scalatest.{FlatSpec, Matchers}

class ParserTest extends FlatSpec with Matchers with
    GeneratorDrivenPropertyChecks with
    TableDrivenPropertyChecks {

  val varNameA ="a"
  val b ="b"
  val test = "test"
  val tests = Table(("input", "output"),
                    ("(!a) and !b", Some(And(Not(Variable(
   varNameA)), Not(Variable(b))))),
                    ("(!a) and b", Some(And(Not(Variable(
   varNameA)), Variable(b)))),
                    ("a and !b", Some(And(Variable(varNameA
   ), Not(Variable(b))))),
                    ("a and (!b)", Some(And(Variable(
   varNameA), Not(Variable(b))))),
                    ("a and b", Some(And(Variable(varNameA)
   , Variable(b)))),
                    ("0", Some(Constant(false))),
                    ("1", Some(Constant(true))),
                    ("", None),
                    ("out", None),
                    ("(", None),
                    (")", None),
                    ("()", None),
                    ("!", None),
                    ("and", None),
                    ("a and", None),
                    ("and a", None),
                    ("or", None),
                    ("a or", None),
                    ("or a", None),
                    ("!a and !b", Some(Not(And(Variable(
varNameA), Not(Variable(b)))))),
                    ("!a and b", Some(Not(And(Variable(
varNameA), Variable(b))))),
                    ("!a or !b", Some(Not(Or(Variable(
varNameA), Not(Variable(b)))))),
                    ("!a or b", Some(Not(Or(Variable(
varNameA), Variable(b))))),
                    ("!a", Some(Not(Variable(varNameA)))),
                    ("(!a)", Some(Not(Variable(varNameA))))
,
                    ("!(a)", Some(Not(Variable(varNameA))))
,
                    ("(!a) or !b", Some(Or(Not(Variable(
varNameA)), Not(Variable(b))))),
                    ("(!a) or b", Some(Or(Not(Variable(
varNameA)), Variable(b)))),
                    ("a or !b", Some(Or(Variable(varNameA),
 Not(Variable(b))))),
                    ("a or (!b)", Some(Or(Variable(varNameA
), Not(Variable(b))))),
                    ("a or b", Some(Or(Variable(varNameA),
Variable(b)))),
                    ("a", Some(Variable(varNameA))),
                    ("(a)", Some(Variable(varNameA))),
                    ("test", Some(Variable(test)))
)

"The table of results" should  "be parsed as expected"
in forAll (tests) {
                (input : String, res : Option[Node]) =>
{
                  Parser.variableParser(input) should be
 (res)
```

```
                Variable.clear()
              }
            }
}
```

Listing 20: PITable.scala

```
package model

import scala.annotation.tailrec

/**
 * Created by joshua on 05/01/15.
 */
object PITable {
  def solve(primeImplicants : List[Implicant], minterms :
     List[Int], vars : List[String]) = {

    val start = new PITable(
      primeImplicants.map(x => new PrimeImplicant(x, x.
     terms().toSet)),
      minterms.toSet,
      Set[Implicant](),
      vars
    )

    bestSolution(start)
  }

  //@tailrec
  def bestSolution(t : PITable) : PITable = t.finished
     match {
    case true => t
    case false => {
      val branches = for (row <- t.rows) yield bestSolution
     (reduceTable(t.selectRow(row)))
      branches.minBy(_.cost(t.vars.length))
    }
  }

  @tailrec def reduceTable(t : PITable) : PITable = t.
     selectEssential match {
    case (true, newTable) => reduceTable(newTable.
     reduceRows)
    case (false, _) => t.reduceRows
  }
}

case class PITable(val rows : List[PrimeImplicant], val
     cols : Set[Int], val results : Set[Implicant], val
     vars :
List[String]) {

  def cost(order : Int) = {
    results.foldLeft(0) {
      _ + _.cost(order)
    }
  }

  def finished = cols.size == 0

  def selectRow(row : PrimeImplicant) = {
    val nRows = rows.filter(_ != row).map(_.reduce(row.
     terms))
    val nCols = cols -- row.terms
    val nRes = results + row.implicant
    this.copy(rows = nRows, cols = nCols, results = nRes)
  }

  def reduceRows = {
    var nRows = rows.filter(!_.empty())
    for (List(a, b) <- rows.combinations(2)) {
      if (a dominates b) {
        nRows = nRows.filter(_ != b)
      } else if (b dominates a) {
        nRows = nRows.filter(_ != a)
      }
    }
    this.copy(rows = nRows)
  }

  def rowsForMinterm(m : Int) = (for (row <- rows if row.
     covers(m)) yield row).filter(_ !=())
```

```
  def selectEssential = {
    var newTable = this
    var done = false
    var effective = false

    while (!done) {
      done = true
      for (m <- cols) {
        newTable.rowsForMinterm(m) match {
          case List(x : PrimeImplicant) => {
            done = false
            effective = true
            newTable = newTable.selectRow(x)
          }
          case _ => ()
        }
      }
    }

    (effective, newTable)
  }

  def toSumOfProducts(vars : List[String]) = {
    assert(finished)

    results.map(_.withVars(vars)).toList.sorted.foldLeft("")
     )((l, r) => if (l != "") "(%s) or (%s)".format(l, r)
     else r)
  }
}
```

Listing 21: Potential.scala

```
package model

trait Potential {
  def isHigh : Boolean
}
```

Listing 22: PrimeImplicant.scala

```
package model

/**
 * Created by joshua on 05/01/15.
 */
//--- class that slowly covers less and less
class PrimeImplicant(val implicant : Implicant, val terms :
     Set[Int]) {
  val tag = implicant.tag
  val order = implicant.order()

  //--- remove given terms from this data's effect
  def reduce(coveredTerms : Set[Int]) = {
    new PrimeImplicant(implicant, terms -- coveredTerms)
  }

  //--- if this is higher order and contains all the
     minterms of the other
  def dominates(other : PrimeImplicant) = ((other.order <=
     order) && (other.terms.subsetOf(terms)))

  //--- whether this contains a given minterm or not
  def covers(minterm : Int) = terms.contains(minterm)

  //--- whether this is now empty
  def empty() = terms.size == 0

  override def equals(that : Any) = that match {
    case other : PrimeImplicant => hashCode == other.
     hashCode
    case _ => false
  }

  //override def hashCode = implicant.hashCode
  override def toString() = terms.mkString("", ",", "")
}
```

Listing 23: PTrans.scala

```
package model
```

```
/**
 * P(g, s, d) = !g -> (s <-> d)
 *
 * P gates can only carry high potential from the source to
     their drain
 */
case class PTrans(input : Node, drain : Wire, source : Wire
    ) extends Transistor {
  override def get : Potential = if (!input.get) source.get
     () else Undriven()
  var drawnGate : Option[AnyRef] = None

  def drainDriven = input.get && (source.get == High())

  override def resetDrawnGates() : Unit = {
    drawnGate = None
    source.resetDrawnGates()
  }
}
```

Listing 24: qmm.scala

```
// Based on github.com/jjfiv/qmm-scala

package model

import helper.Parser

import scala.annotation.tailrec
import scala.collection.immutable.List

object qmm {
  def main(args : Array[String]) {
    println(method(List(0, 1, 2, 3, 4, 7, 6, 8, 11, 13, 15)
     , Nil, letters("ABCD")))
    println(method(List(1, 2, 3, 5, 9, 10, 11, 18, 19, 20,
     21, 23, 25, 26, 27), Nil, letters("ABCDE")))
    //--- cyclic:
    println(method(List(8, 10, 16, 18, 19, 20, 21, 23, 25,
     27, 29, 40, 42, 43, 46, 47, 55), Nil, letters("ABCDEF")
     ))

    //--- don't cares
    println(method(List(1, 4, 7, 14, 17, 20, 21, 22, 23),
     List(0, 3, 6, 19, 30), letters("ABCDE")))
    //method(List(0,1,2,3),Nil,List("X","Y"))

    //method(List(0,1,2,3), Nil, letters("AB"))
    val res = method(List(1, 2), Nil, letters("AB"))
    method(List(0, 1, 2, 3), Nil, letters("AB"))
    println(Parser.variableParser(res))

    ()
  }

  def method(minterms : List[Int], dontcares : List[Int],
     vars : List[String]) : String = {
    val implicants = (minterms ::: dontcares).sorted.sortBy
     (bitcount(_)).map(new Implicant(_))
    val order = vars.length

    val prime_implicants = genImplicants(implicants, order).
     filter(_.prime)

    val results = PITable.solve(prime_implicants, minterms,
      vars)

    val res = results.toSumOfProducts(vars)

    return res
  }

  def bitcount(x : Int) : Int = {
    @tailrec def bcrec(accum : Int, n : Int) : Int = n
     match {
      case 0 => accum
      case x => bcrec(accum + (x % 2), (x >>> 1))
    }
    bcrec(0, x)
  }

  def genImplicants(zero_cubes : List[Implicant], order :
```

```
      Int) : List[Implicant] = {

    import scala.collection.mutable.{Set => MutableSet}

    //--- generate list and populate with zero-cubes
    var implicants = MutableSet[Implicant]()
    for (i <- zero_cubes) implicants += i

    //--- operate on current order until highest reached
    for (currentOrder <- 0 until order) {
      //--- grab all implicants of the current order
      val data = implicants.toList.filter(_.order ==
     currentOrder)

      for (List(a, b) <- data.combinations(2)) {
        if (a.canCombine(b)) {
          a.prime = false
          b.prime = false

          val n = a.combine(b)

          implicants += n
        }
      }
    }

    implicants.toList
  }

  def letters(x : String) : List[String] = x.split("").
     filter(_ != "").toList
}
```

Listing 25: QuineMcCluskeyTest.scala

```
package model.test

import helper.Parser
import model.{And, LogicalFunction, Variable}
import org.scalatest.{FlatSpec, Matchers}

/**
 * Created by joshua on 04/02/15.
 */
class QuineMcCluskeyTest extends FlatSpec with Matchers {
  private val test : String = "(!a and b and !c and !d) or
     (a and !b and !c and !d) or (a and !b and c and !d) or
      (a " +
    "and !b and c and d) or (a and b and !c and !d) or (a
     and b and c and d)"

  "a and b" should "be reduced to Some(And(a), And(b))" in
     {
    val expr = Parser.variableParser("a and b") match {
      case Some(x) => x
    }
    LogicalFunction.quineMcCluskey(expr) should be(Some(
      And(Variable("a"), Variable("b"))
    )
    )
  }

  test should "be parsed as Some(...)" in {
    LogicalFunction.quineMcCluskey(Parser.variableParser(
     test) match { case Some(x) => x }) should be(Some(
      Variable("x")
    )
    )
    Variable.clear()
  }
}
```

Listing 26: Result.scala

```
package model

object Result extends Wire with Input with Output {
  override def removeGate (transistor : Transistor) : Unit
     = {
    sources = sources diff Array(transistor)
    drains = drains diff Array(transistor)
  }
```

```
  override def get() : Potential = {
    var result : Potential = Undriven()
    for (gate <- getSources) {
      if (gate.get != Undriven()) {
        result = gate.get
      }
    }
    if (result == Undriven()) {
      for (gate <- getDrains) {
        if (gate.get != Undriven()) {
          result = gate.get
        }
      }
    }
    if (result == Undriven()) {
      result//throw new RuntimeException("Result not driven
     ")
    } else {
      result
    }
  }

  override def clear : Unit = {
    clearDrains
    clearSources
    Source.clear
    Drain.clear
  }

  override def toString() = "Result"

  override def resetDrawnGates() : Unit = {
    for (gate <- getSources) {
      gate.resetDrawnGates()
    }
    for (gate <- getDrains) {
      gate.resetDrawnGates()
    }
    ()
  }
}
```

Listing 27: Source.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
object Source extends Wire with Output {
  override def removeGate (transistor : Transistor) : Unit
     = {
    drains = drains diff Array(transistor)
  }

  def get() : Potential = High()

  override def clear : Unit = clearDrains

  override def toString() = "Source"

  override def resetDrawnGates () : Unit = ()
}
```

Listing 28: Transistor.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
abstract class Transistor {
  def remove () = {
    drain.removeGate(this)
    source.removeGate(this)
  }

  val input : Node
  val drain : Wire
  val source : Wire
  var drawnGate : Option[AnyRef]
  def resetDrawnGates() : Unit
```

```
  def get() : Potential
}
```

Listing 29: Undriven.scala

```
package model

/**
 * Created by joshua on 17/12/14.
 */
case class Undriven() extends Potential {
  override def isHigh : Boolean = false
}
```

Listing 30: Variable.scala

```
package model

import scala.collection.immutable.TreeMap

case class Variable(val ident : String) extends Node with
    Atom {
  def get = Variable.lookup(ident)

  override def toString = ident
}

object Variable {
  def negateAll(expr : Node) : Unit = {
    for (variable <- identMap) {
      setValue(variable._1, false)
    }
    setValue("out", expr.get())
  }

  private var identMap = new TreeMap[String, Boolean]
  identMap += Tuple2("out", false)
  private var intermediate = 0;

  def create(ident : String, value : Boolean) : Variable =
    {
    if (!(identMap contains ident))
      identMap += Tuple2(ident, value)
    Variable(ident)
  }

  def lookup(ident : String) = identMap apply ident

  def setValue(ident : String, value : Boolean) = identMap
    += Tuple2(ident, value)

  def clear() = {
    identMap = new TreeMap[String, Boolean]
    identMap += Tuple2("out", false)
    intermediate = 0
  }

  def getMap = identMap
}
```

Listing 31: WireHigh.scala

```
package model

/**
 * Created by joshua on 17/04/15.
 */
class WireHigh extends Wire with Input with Output {
  override def removeGate (transistor : Transistor) : Unit
    = {
    sources = sources filter (x => !(x eq transistor))
    drains = drains filter (x => !(x eq transistor))
  }

  override def get() : Potential = {
    var res : Potential = Undriven()
    for (source <- getSources) {
      if (source.get != Undriven()) {
        res = source.get
      }
```

```
    }
    res
  }

  override def clear : Unit = {
    clearSources
    clearDrains
  }

  override def toString() = ""

  override def resetDrawnGates() : Unit = {
    for (source <- getSources) {
      source.resetDrawnGates()
    }
  }
}
```

Listing 32: WireLow.scala

```
package model

/**
 * Created by joshua on 17/04/15.
 */
class WireLow extends Wire with Input with Output {
  override def removeGate (transistor : Transistor) : Unit
     = {
    sources = sources filter (x => !(x eq transistor))
    drains = drains filter (x => !(x eq transistor))
  }

  override def get() : Potential = {
    var res : Potential = Undriven()
    for (drain <- getDrains) {
      if (drain.get != Undriven()) {
        res = drain.get
      }
    }
    res
  }

  override def clear : Unit = {
    clearSources
    clearDrains
  }

  override def toString() = ""

  override def resetDrawnGates() : Unit = {
    for (drain <- getDrains) {
      drain.resetDrawnGates()
    }
  }
}
```

Listing 33: Wire.scala

```
package model

abstract class Wire {
  def removeGate (transistor : Transistor) : Unit

  protected var sources = Array[Transistor]()
  protected var drains = Array[Transistor]()

  def get() : Potential

  def addSource(node : Transistor) = {
    sources = node +: sources
  }

  def getSources = sources

  def clearSources : Unit = {
    sources = Array[Transistor]()
  }

  def clear() : Unit

  def getDrains = drains
```

```
  def addDrain(node : Transistor) = {
    drains = node +: drains
  }

  def clearDrains : Unit = {
    drains = Array[Transistor]()
  }

  def resetDrawnGates() : Unit
}
```

A.2 View

Listing 34: DrawCircuit.scala

```
package view

import com.mxgraph.util.mxConstants
import com.mxgraph.view.mxGraph
import model._

/**
 * Created by joshua on 08/04/15.
 */
case class DrawCircuit (val graph : mxGraph) {
  val nodes = scala.collection.mutable.Stack[AnyRef]()
  val deltaY = 50
  val deltaX = 25
  val lastSources = scala.collection.mutable.MutableList[
    AnyRef]()
  val lastDrains = scala.collection.mutable.MutableList[
    AnyRef]()
  var parent = graph.getDefaultParent
  var currentX = 0;
  var sourcesYLimit = 0;
  var drainsYLimit = 0;
  var maxX = 0;

def draw () : Unit = {
  parent = graph.getDefaultParent
  // No gates are currently drawn, so reset
  Result.resetDrawnGates()
  styleSheet("nmos_de")
  styleSheet("pmos_de")
  styleSheet("nmos_en")
  styleSheet("pmos_en")
  wireStyle(true)
  wireStyle(false)
  otherWireStyle()
  graph.getModel.beginUpdate()
  try { {
    val node = graph.insertVertex(parent, null, "output",
    0, 0, 0, 1)
    val sources = Result.getSources
    // every gate in sources should connect to the output
    wire
    for (sources <- sources) {
      nodes push node
    }
    currentX = 0
    drawNetwork(sources, 0, true)
    maxX = currentX
    currentX = -10
    val drains = Result.getDrains
    for (drain <- drains) {
      nodes push node
    }
    drawNetwork(drains, 0, false)
    maxX = Math.max(maxX, currentX)
    val newOut = graph.insertVertex(parent, null, "output
  ", -15, 0, maxX + 30, 1, "wireUndriven")

    val drain = graph.insertVertex(parent, null, "drain",
    -15, drainsYLimit, maxX + 30, 1,
                                    "wireDriven")

    val source = graph.insertVertex(parent, null, "source
  ", -15, -1.0 * sourcesYLimit, maxX + 30, 1,
                                     "wireDriven")
```

```
      if (Result.get() == High()) {
        graph.setCellStyle("wireDriven", Array(newOut))
      }

      for (edge <- graph.getEdges(node)) {
        graph.splitEdge(edge, Array(newOut))
      }

      for (sourceGate <- lastSources) {
        val x = graph.getCellGeometry(sourceGate).
    getCenterX
        val dot = graph.insertVertex(parent, null, "", x,
    -1.0 * sourcesYLimit, 1, 1)
        graph.insertEdge(parent, null, "", sourceGate, dot,
     "wireOther")
      }

      for (drainGate <- lastDrains) {
        val x = graph.getCellGeometry(drainGate).getCenterX
        val dot = graph.insertVertex(parent, null, "", x,
    drainsYLimit, 1, 1)
        graph.insertEdge(parent, null, "", drainGate, dot,
    "wireOther")
      }

      lastSources.clear()
      lastDrains.clear()

      graph.removeCells(Array(node))
    }
    } finally {
      graph.getModel.endUpdate()
    }
  }

  private def styleSheet (name : String) : Unit = {
    // add gate as node at (xPos, y+1)
    // get graph stylesheet
    val stylesheet = graph.getStylesheet

    // define image style name
    val styleName = name

    // define image style
    val style = new java.util.HashMap[String, AnyRef]()
    style.put(mxConstants.STYLE_SHAPE, mxConstants.
     SHAPE_IMAGE)
    style.put(mxConstants.STYLE_IMAGE, "file:resources/%s.
     png".format(name))
    style.put(mxConstants.STYLE_LABEL_POSITION, mxConstants.
     ALIGN_LEFT)

    stylesheet.putCellStyle(styleName, style)
  }

  private def otherWireStyle() : Unit = {
    // get graph stylesheet
    val stylesheet = graph.getStylesheet

    // define image style name
    val styleName = "wireOther"

    // define image style
    val style = new java.util.HashMap[String, AnyRef]()
    style.put(mxConstants.STYLE_ENDARROW, mxConstants.NONE)
    style.put(mxConstants.STYLE_STARTARROW, mxConstants.
     NONE)
    style.put(mxConstants.STYLE_STROKECOLOR, "red")
    style.put(mxConstants.STYLE_FILLCOLOR, "red")
    style.put(mxConstants.STYLE_LABEL_POSITION, mxConstants.
     ALIGN_RIGHT)

    stylesheet.putCellStyle(styleName, style)

  }

  private def wireStyle (driven : Boolean) : Unit = {
    // get graph stylesheet
    val stylesheet = graph.getStylesheet

    // define image style name
    val styleName = "wire" + (if (driven) {
      "Driven"
```

```
      } else {
        "Undriven"
      })

      // define image style
      val style = new java.util.HashMap[String, AnyRef]()
      style.put(mxConstants.STYLE_SHAPE, mxConstants.
       SHAPE_LINE)
      style.put(mxConstants.STYLE_STROKECOLOR, if (driven) {
        "red"
      } else {
        "black"
      })
      style.put(mxConstants.STYLE_FILLCOLOR, if (driven) {
        "red"
      } else {
        "black"
      })
      style.put(mxConstants.STYLE_LABEL_POSITION, mxConstants.
       ALIGN_RIGHT)

      stylesheet.putCellStyle(styleName, style)
    }

    private[this] def drawNetwork (gates : Array[Transistor],
       y : Int, drawingTopNetwork : Boolean) : Unit = {
      if (gates.isEmpty) {
        // if last node on the chain, nothing to connect it
       to, so forget it
        if (drawingTopNetwork) {
          lastSources += nodes.pop()
          sourcesYLimit = scala.math.max(sourcesYLimit, y)
        } else {
          lastDrains += nodes.pop()
          drainsYLimit = scala.math.max(drainsYLimit, y)
        }
      } else {
        for (gate <- gates) {
          gate.drawnGate match {
            case Some(node : AnyRef) => {
              val box = graph.getCellGeometry(node)
              val x = box.getCenterX - deltaX * 2
              val tempy = box.getCenterY + (box.getHeight * (
    if (drawingTopNetwork) {
                -0.5
              } else {
                0.5
              })) - deltaY
              val point = graph insertVertex(parent, null, ""
    , x, tempy, 0, 0)
              graph.insertEdge(parent, null, "", nodes.pop(),
     node,
                               mxConstants.STYLE_SHAPE + "="
    + mxConstants.SHAPE_LINE)
            }
            case None => {
              val previousNode = nodes.pop()
              val node =
                graph.insertVertex(parent, null, gate.input.
    toString, currentX, if (drawingTopNetwork) {
                  -(y + deltaY)
                } else {
                  y
                }, deltaX * 2.0, deltaY, (if (!
    drawingTopNetwork) {
                  "nmos"
                } else {
                  "pmos"
                }) + (if (gate.get() == Undriven()) {
                  "_de"
                } else {
                  "_en"
                }))
              gate.drawnGate = Some(node)
              graph insertEdge
              (parent, null, "", previousNode, node,
    mxConstants.STYLE_SHAPE + "=" + mxConstants.
    SHAPE_IMAGE)
              nodes push node
              drawNetwork(if (drawingTopNetwork) {
                gate.source.getSources
              } else {
                gate.drain.getDrains
              }, y + deltaY, drawingTopNetwork)
```

```
            currentX += deltaX * 2
          }
        }
      }
    }
  }
}
```

A.3 Controller

Listing 35: Gui.java

```
package controller;

import com.mxgraph.io.mxCodec;
import com.mxgraph.layout.mxGraphLayout;
import com.mxgraph.layout.mxParallelEdgeLayout;
import com.mxgraph.model.mxGraphModel;
import com.mxgraph.swing.mxGraphComponent;
import com.mxgraph.util.mxCellRenderer;
import com.mxgraph.util.mxXmlUtils;
import com.mxgraph.util.png.mxPngEncodeParam;
import com.mxgraph.util.png.mxPngImageEncoder;
import com.mxgraph.view.mxGraph;
import com.mxgraph.view.mxGraphView;
import helper.CMOSLayout;
import helper.Parser;
import model.Node;
import model.Result;
import model.Variable;
import scala.Option;
import scala.Tuple2;
import scala.collection.Iterator;
import scala.collection.immutable.TreeMap;
import view.DrawCircuit;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileOutputStream;
import java.net.URLEncoder;
import java.util.HashMap;

/**
 * Created by joshua on 08/04/15.
 */
public class Gui {
    /**
     * Handler for the whole graph.
     */
    public mxGraph graph;
    /**
     * Handler for the graphic component.
     */
    public mxGraphComponent graphComponent = null;
    /**
     * A handler for the graph's layout.
     */
    public mxGraphLayout layout;
    private JPanel mainPanel;
    private JPanel visPanel;
    private JPanel inputPanel;
    private JTextField textInput;
    private JButton goButton;
    private JCheckBox outputCheckBox;
    private JPanel variableHolder;
    private JButton exportButton;
    private HashMap<String, JCheckBox> nameBoxMap = new
     HashMap<String, JCheckBox>();
    private DrawCircuit drawCircuit;

    public Gui () {
        goButton.addActionListener(new ActionListener() {
            @Override public void actionPerformed (
     ActionEvent actionEvent) {
```

```
            // Clear checkboxes
            variableHolder.removeAll();
            variableHolder.add(outputCheckBox);
            Variable.clear();

            Option<Node> parseResult = Parser.
variableParser(textInput.getText());

            if (parseResult.isEmpty()) {
                JOptionPane.showMessageDialog(mainPanel
, "Could not parse expression", "Error",
JOptionPane.ERROR_MESSAGE);
            } else {
                final Node result = parseResult.get();
                String message = CMOSLayout.layout(
result);
                JOptionPane.showMessageDialog(mainPanel
, message, "Finished", JOptionPane.INFORMATION_MESSAGE)
;
                final TreeMap<String, Object> map =
Variable.getMap();
                final Iterator<Tuple2<String, Object>>
it = map.iterator();
                visPanel.setSize(400, 300);
                if (graph != null) {
                    ((mxGraphModel) graph.getModel()).
clear();
                }
                graphComponent = initGraph();
                visPanel.add(graphComponent,
BorderLayout.CENTER);
                mainPanel.updateUI();
                mainPanel.validate();
                mainPanel.repaint();
                while (it.hasNext()) {
                    Tuple2<String, Object> value = it.
next();
                    String variableName = value._1();
                    final JCheckBox variableBox = new
JCheckBox(variableName, false);
                    if (!variableName.equals("out")) {
                        variableBox.addItemListener(new
 ItemListener() {
                            @Override public void
itemStateChanged (ItemEvent e) {
                                Variable.setValue(
variableBox.getText(), variableBox.isSelected());
                                outputCheckBox.
setSelected(Result.get().isHigh());
                                // Clear Graph
                                if (graph != null) {
                                    ((mxGraphModel)
graph.getModel()).clear();
                                }
                                drawCircuit.draw();

                                resizeGraphView(
graphComponent);
                            }
                        });
                        variableBox.setSelected(
Variable.lookup(variableName));
                        variableHolder.add(variableBox)
;
                        nameBoxMap.put(variableName,
variableBox);
                        variableHolder.validate();
                        variableHolder.repaint();
                    }
                }
            }
            outputCheckBox.setSelected(parseResult.get
().get());
        }
    });
    exportButton.addActionListener(new ActionListener()
 {
        @Override public void actionPerformed (
ActionEvent actionEvent) {
            try {
                JFileChooser fileChooser = new
JFileChooser();
```

```
                if (fileChooser.showSaveDialog(
        mainPanel) == JFileChooser.APPROVE_OPTION) {
                    File file = fileChooser.
        getSelectedFile();
                    String filename = file.
        getAbsolutePath();

                    BufferedImage image =
        mxCellRenderer.createBufferedImage(graph, null, 1,
        Color.WHITE,

                      graphComponent.isAntiAlias(),
        null,

                      graphComponent.getCanvas());
                    mxPngEncodeParam param = null;
                    if (image != null) {
                        param = mxPngEncodeParam.
        getDefaultEncodeParam(image);
                    } else {
                        JOptionPane.showMessageDialog(
        mainPanel, "Could not create image", "Error",
        JOptionPane

                                                      .
        ERROR_MESSAGE);
                    }

                    // Creates the URL-encoded XML data
                    mxCodec codec = new mxCodec();
                    String xml = URLEncoder.encode(
        mxXmlUtils.getXml(codec.encode(graph.getModel())), "
        UTF-8");

                    param.setCompressedText(new String
        [] { "graph", xml });

                    // Saves as a PNG file
                    FileOutputStream outputStream = new
         FileOutputStream(new File(filename));

                    try {
                        mxPngImageEncoder encoder = new
         mxPngImageEncoder(outputStream, param);

                        if (image != null) {
                            encoder.encode(image);
                        } else {
                            System.out.println("No
        Image");
                        }
                    } finally {
                        outputStream.close();
                    }
                }
            } catch (Exception e) {

            }
        }
    });
}

public static void main (String[] args) {
    try {
        UIManager.setLookAndFeel(UIManager.
     getSystemLookAndFeelClassName());
    } catch (Exception e1) {
        e1.printStackTrace();
    }

    JFrame frame = new JFrame("CMOS Calculator");
    frame.setContentPane(new Gui().mainPanel);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
     );
    frame.pack();
    frame.setVisible(true);
}

/**
 * This will initialize the graph component to draw the
  shapes on.
 *
 * @return The graph component which was drawn on.
 */
```

```
public final mxGraphComponent initGraph () {
    mxGraphComponent localGraphComponent = null;
    try {
        graph = new mxGraph();
        localGraphComponent = new mxGraphComponent(
graph);

        // Allow negative co-ordinates (makes drawing
easier as bottom half can be negatively positioned)
        graph.setAllowNegativeCoordinates(true);
        // dangling edges are bad and result in all
kinds of nasty things
        graph.setAllowDanglingEdges(false);
        // edge source and target are the same
        graph.setAllowLoops(true);
        // don't need this
        graph.setCellsResizable(false);
        // don't allow movement
        graph.setCellsMovable(false);
        // don't allow new connections
        graph.setConnectableEdges(false);
        // make editing labels more comfortable
        localGraphComponent.setEnterStopsCellEditing(
true);
        // antialiasing \o/
        localGraphComponent.setAntiAlias(true);
        // set size
        localGraphComponent.setSize(visPanel.getWidth()
, visPanel.getHeight());

        graph.setAutoOrigin(true);

        // define a parallel layout for the edges
        layout = new mxParallelEdgeLayout(graph);
        drawCircuit = new DrawCircuit(graph);
        drawCircuit.draw();

        resizeGraphView(localGraphComponent);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(mainPanel, e.
getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    } finally {
        return localGraphComponent;
    }
}

private void resizeGraphView (mxGraphComponent
 localGraphComponent) {
    mxGraphView view = localGraphComponent.getGraph().
 getView();
    int compLenH = localGraphComponent.getHeight();
    int viewLenH = (int) view.getGraphBounds().
 getHeight();
    double scaleH = (double) (compLenH - 1) / viewLenH
 * view.getScale();

    int compLenW = localGraphComponent.getWidth();
    int viewLenW = (int) view.getGraphBounds().getWidth
 ();

    double scaleW = (double) (compLenW - 1) / viewLenW
 * view.getScale();

    view.setScale(Math.min(scaleH, scaleW));
}

{
    // GUI initializer generated by IntelliJ IDEA GUI
 Designer
    // >>> IMPORTANT!! <<<
    // DO NOT EDIT OR ADD ANY CODE HERE!
    setupUI();
}

/**
 * Method generated by IntelliJ IDEA GUI Designer >>>
 IMPORTANT!! <<< DO NOT edit this method OR call it in
 your
 * code!
 *
 * @noinspection ALL
 */
private void setupUI () {
    mainPanel = new JPanel();
```

```
        mainPanel.setLayout(new BorderLayout(0, 0));
        visPanel = new JPanel();
        visPanel.setLayout(new BorderLayout(0, 0));
        visPanel.setMinimumSize(new Dimension(400, 300));
        visPanel.setPreferredSize(new Dimension(400, 300));
        mainPanel.add(visPanel, BorderLayout.CENTER);
        visPanel.setBorder(BorderFactory.createTitledBorder
("CMOS Output"));
        inputPanel = new JPanel();
        inputPanel.setLayout(new BorderLayout(0, 0));
        mainPanel.add(inputPanel, BorderLayout.WEST);
        final JPanel panel1 = new JPanel();
        panel1.setLayout(new FlowLayout(FlowLayout.CENTER,
5, 5));
        panel1.setEnabled(true);
        inputPanel.add(panel1, BorderLayout.NORTH);
        textInput = new JTextField();
        textInput.setMinimumSize(new Dimension(70, 26));
        textInput.setPreferredSize(new Dimension(100, 26));
        textInput.setText("");
        textInput.setToolTipText("Enter the text to parse
here");
        panel1.add(textInput);
        goButton = new JButton();
        goButton.setText("Parse!");
        panel1.add(goButton);
        final JScrollPane scrollPane1 = new JScrollPane();
        inputPanel.add(scrollPane1, BorderLayout.CENTER);
        variableHolder = new JPanel();
        variableHolder.setLayout(new FlowLayout(FlowLayout.
CENTER, 5, 5));
        scrollPane1.setViewportView(variableHolder);
        outputCheckBox = new JCheckBox();
        outputCheckBox.setActionCommand("");
        outputCheckBox.setEnabled(false);
        outputCheckBox.setText("out");
        variableHolder.add(outputCheckBox);
        final JPanel panel2 = new JPanel();
        panel2.setLayout(new BorderLayout(0, 0));
        inputPanel.add(panel2, BorderLayout.SOUTH);
        exportButton = new JButton();
        exportButton.setText("Export");
        panel2.add(exportButton, BorderLayout.NORTH);
    }

    /** @noinspection ALL */
    public JComponent getRootComponent() {
        return mainPanel;
    }
}
```

A.4 Helper

Listing 36: CMOSLayout.scala

```
package helper

import model._

import scala.collection.mutable
import scala.collection.mutable.{HashSet, Stack}

/**
 * Created by joshua on 08/01/15.
 */
object CMOSLayout {

  private var totalGates = 0

  def main(args: Array[String]): Unit = {
    layout(helper.Parser.variableParser("(x and y) or ((!x)
      and (!y) and z)") match {
       case Some(v) => {
         println(v); v
       }
    })
    println(Result.getSources.length)
    for (source <- Result.getSources) println(source.
     toString)
    println(Result.getDrains.length)
```

```
  for (drain <- Result.getDrains) println(drain.toString)
}

private val negations = new HashSet[Variable]()

def checkForDuplicateGates (wire : Wire)(
   checkingTopNetwork : Boolean) : Unit  = {
  // if we're at the result, we've finished
  if (wire != Result) {
    val wiresToCheck = new mutable.HashSet[Wire]()
    val gates = new mutable.HashMap[Node, Transistor]()
    for (gate <- (if(checkingTopNetwork) wire.getDrains
   else wire.getSources)) {
      if (gates.contains(gate.input)) {
        if (checkingTopNetwork) {
          for (drain <- gate.drain.getDrains) gates(gate.
   input).drain.addDrain(drain)
          wiresToCheck += gates(gate.input).drain
        } else {
          for (source <- gate.source.getSources) gates(
   gate.input).source.addSource(source)
          wiresToCheck += gates(gate.input).source
        }
        gate.remove()
        totalGates -= 1
      } else {
        gates.put(gate.input, gate)
      }
    }
    for (wire <- wiresToCheck) checkForDuplicateGates(
   wire)(checkingTopNetwork)
  }
}

def layout(expr: model.Node): String = {
  val normal = tryLayout(expr)
  val doubleNegate = tryLayout(Not(expr)) + 2

  if (normal <= doubleNegate) {
    tryLayout(expr)
    return normal + "transistors used"
  } else {
    return doubleNegate + "transistors used, with two to
   negate the output"
  }
}

def tryLayout(expr : model.Node) : Integer = {
  Result.clear;
  totalGates = 0;
  negations.clear()

  LogicalFunction.quineMcCluskey(expr) match {
    case Some(x) => {
      execute(x)(true)
    }
    case None => {
      throw new RuntimeException("Expression not
   minimised correctly")
    }
  }
  LogicalFunction.quineMcCluskey(Not(expr)) match {
    case Some(x) => {
      execute(x)(false)
    }
    case None => {
      throw new RuntimeException("Expression not
   minimised correctly")
    }
  }
  Variable.negateAll(expr)
  //checkForDuplicateGates(Source)(true)
  //checkForDuplicateGates(Drain)(false)
  (totalGates + 2 * negations.size)
}

/**
 * pre: expr has been converted to sum of products form (
   DNF)
 * @param expr - the expression to be implemented in Gui
 * @param buildingTopNetwork - indicates if the network
   we are building is to be used to carry a high
   potential or
 *                            a low potential
```

```
 * @return - side effects on the Result object
 */
private def execute(expr: model.Node)(buildingTopNetwork:
    Boolean): Unit = {
  // I : \/ stack or subExpr = expr
  // Initially stack = [], subExpr = expr => I
  val stack = Stack[model.Node]()
  var subExpr = expr

  while (subExpr != Constant(false)) {
    subExpr match {
      case Or(x, y) => {
        stack push y
        subExpr = x
      }
      case Variable(_) => {
        println("Processing: " + subExpr.toString)
        convertToGates(subExpr, buildingTopNetwork)
        if (!stack.isEmpty) {
          subExpr = stack.pop()
        } else {
          subExpr = Constant(false)
        }
      }
      case Not(_) => {
        println("Processing: " + subExpr.toString)
        convertToGates(subExpr, buildingTopNetwork)
        if (!stack.isEmpty) {
          subExpr = stack.pop()
        } else {
          subExpr = Constant(false)
        }
      }
      case And(_, _) => {
        println("Processing: " + subExpr.toString)
        convertToGates(subExpr, buildingTopNetwork)
        if (!stack.isEmpty) {
          subExpr = stack.pop()
        } else {
          subExpr = Constant(false)
        }
      }
    }
  }
}

/**
 * As the parser operates using a fold, the right hand
   node must always be an Or(x,y), with the last node
   being made
 * up of a Variable(x) or Not(Variable(x))
 *
 * For some expression, assuming buildingTopNetwork,
   x_0 ∧ x_1 ∧ ... ∧ x_n, with literals x_0, x_1, ... x_n, x_0's
 * drain is attached to the output wire, x_0's source is
   attached to a new wire, which is then attached to x_1's
 * drain, working along the conjunction until the last
   node.  x_n is attached to the source wire.
 *
 * If not buildingTopNetwork, replace all instances of
   source with drain, and attach x_n to the drain wire.
 *
 * @param node the logical expression to be converted to
   a series of gates
 * @param buildingTopNetwork indicates if the top half,
   or bottom is being evaluated
 *                             (fromBottom => PGate, !
   fromBottom => NGate)
 *
 *                             If a connection is made
   above a wire, then a source is being added, and if it
   is being made below a wire, then a
 *                             drain is added
 */
private def convertToGates(node: Node, buildingTopNetwork
  : Boolean) = {
  var subNode: model.Node = node
  var currentWire: Wire = Result

  while (subNode != Constant(false)) subNode match {
    case And(x, y) => {
      val wire = if (buildingTopNetwork) { new WireHigh()
    } else { new WireLow() }
      x match {
```

```
      case Not(Variable(v)) => {
        if (buildingTopNetwork) {
          val gate = new PTrans(Variable(v),
currentWire, wire)
          currentWire addSource gate
          wire addDrain gate
          println("Adding source to " + currentWire.
toString())
        } else {
          val gate = new NTrans(Not(Variable(v)), wire,
 currentWire)
          negations add Variable(v)
          currentWire addDrain gate
          wire addSource gate
          println("Adding drain to " + currentWire.
toString())
        }
      }
      case Variable(v) => {
        if (buildingTopNetwork) {
          val gate = new PTrans(Not(Variable(v)),
currentWire, wire)
          negations add Variable(v)
          currentWire addSource gate
          wire addDrain gate
          println("Adding source to " + currentWire.
toString())
        } else {
          val gate = new NTrans(Variable(v), wire,
currentWire)
          currentWire addDrain gate
          wire addSource gate
          println("Adding drain to " + currentWire.
toString())
        }
      }
    }
    currentWire = wire
    subNode = y
  }
  // Create a new gate and finish
  case Not(Variable(v)) => {
    if (buildingTopNetwork) {
      val gate = new PTrans(Variable(v), currentWire,
Source)
      currentWire addSource gate
      Source addDrain gate
      println("Adding source to " + currentWire.
toString())
    } else {
      val gate = new NTrans(Not(Variable(v)), Drain,
currentWire)
      negations add Variable(v)
      currentWire addDrain gate
      Drain addSource gate
      println("Adding drain to " + currentWire.toString
())
    }
    subNode = Constant(false)
  }
  case Variable(v) => {
    if (buildingTopNetwork) {
      val gate = new PTrans(Not(Variable(v)),
currentWire, Source)
      negations add Variable(v)
      currentWire addSource gate
      Source addDrain gate
      println("Adding source to " + currentWire.
toString())
    } else {
      val gate = new NTrans(Variable(v), Drain,
currentWire)
      currentWire addDrain gate
      Drain addSource gate
      println("Adding drain to " + currentWire.toString
())
    }
    subNode = Constant(false)
  }
 }
}

private def exprNotAtom(expr: Node) : Boolean = expr
  match {
```

```
    case And(_, _) => true
    case Or(_, _) => true
    case _ : Atom => false
    case _ => true
  }

  def exprNotConjunction(node: Node): Boolean = node match
    {
    case And(_, _) => false
    case _ => true
  }
}
```

Listing 37: LogicalExpression.scala

```
package helper

import model._

import scala.util.parsing.combinator.{PackratParsers,
    RegexParsers}

/**
 * \Sigma = [A-Za-z0-9]*
 * expr -> conj | disj | literal
 * conj -> literal | literal "and" conj
 * disj -> conj | conj "or" disj
 * literal -> "!" simp | simp
 * simp -> variable | constant | "(" expr ")" | expr
 * variable -> \Sigma\Sigma*\{"out", "0", "1"}
 * constant -> 0 | 1
 **/
class LogicalExpression extends RegexParsers with
    PackratParsers {
  lazy val variable : PackratParser[Node] = """^(?!out|and|
    or$)([A-Za-z0-9]+)""".r ^^ {
    v => {
      Variable.create(v, false)
    }
  }

  lazy val const : PackratParser[Node] = ("1" | "0") ^^ {
    case "1" => {
      Constant(true)
    }
    case "0" => {
      Constant(false)
    }
  }

  lazy val literal : PackratParser[Node] = ("!" ~ simp |
    simp) ^^ {
    case "!" ~ (n : Node) => {
      Not(n)
    }
    case n : Node => {
      n
    }
  }

  lazy val conj : PackratParser[Node] = rep1sep(literal, "
    and") ^^ {
    _.reduceRight(And)
  }

  lazy val disj : PackratParser[Node] = rep1sep(conj, "or")
     ^^ {
    _.reduceRight(Or)
  }

  lazy val simp : PackratParser[Node] = (variable
                                       ||| const
                                       ||| ("(" ~ expr ~
    ")" ^^ {
    case "(" ~ e ~ ")" => {
      e
    }
  })
                                       ||| expr)

  lazy val expr : PackratParser[Node] = (conj
                                       ||| disj
                                       ||| literal)
```

```
}
```

Listing 38: Parser.scala

```
package helper

import model.Node

object Parser extends LogicalExpression {

  def variableParser(x : String) : Option[Node] = {
    val result = parseAll(expr, x)
    if (result.successful)
      Some(result.get)
    else
      None
  }
}
```