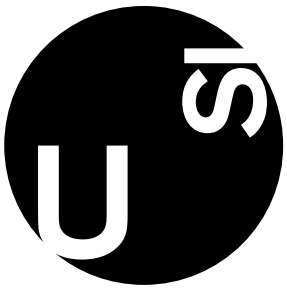

Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

June 24, 2022

SAT-based Techniques for Approximate Circuit Design

Matteo Alberici

Abstract

As energy efficiency becomes a crucial concern in digital applications, Approximate Computing (AC) has gained popularity, since it represents a potential answer to this ever-growing energy quest. AC brings a new perspective on digital circuit design by adding accuracy to the set of design metrics: it allows slight accuracy reduction in exchange for a significant improvement in energy consumption. AC is particularly suited for error-resilient applications, where such minor accuracy losses do not significantly reduce the quality of the result. In particular, we are interested in Approximate Logic Synthesis (ALS), which is the process of automatically generating, given an exact circuit and a tolerated error threshold, an approximate circuit counterpart where the error is assured to be lower than the specified threshold. The resulting circuit will be a functional modification of the original circuit, where some parts will be substituted or perhaps entirely removed. While several design algorithms have been proposed, we investigate innovative SAT-based solutions. In computer science, multiple problems can be reduced to the SAT problem, including the approximate circuit design challenge investigated here.

Advisor
Prof. Laura Pozzi
Assistant
Dr. Ilaria Scarabottolo

Advisor's approval (Prof. Laura Pozzi): Date:

Contents

1 Introduction

1.1 Project Context

1.1.1 Definition of Approximate Computing

Approximate Computing (AC) [1] is an emerging paradigm that improves design area and power consumption by loosening the requirement for total accuracy and, instead, returning potentially inaccurate results rather than guaranteed accurate ones. Since performing exact computations in many scenarios requires an enormous amount of resources, AC [2] provides a way of obtaining gains in both performance and energy consumption, while still achieving results with acceptable precision. The application efficiency is thus improved by leveraging error resilience. AC performs skillfully for applications such as multimedia processing and voice recognition [3], wherein the underlying computation has intrinsic resilience to minor errors. These applications are abundant in different fields such as computer vision, machine learning, and signal processing.

1.1.2 Approximate Circuits

Approximate circuits are approximate computing platforms' fundamental hardware building blocks [4]. Such circuits outperform traditional ones in speed and design area at the cost of a slight loss in computational accuracy. According to a pre-determined set of quality constraints, an approximate circuit is the realization of a logic function that slightly deviates from the original specification.

1.1.3 Automatic Design of Approximate Circuits

In *approximate circuit* design [3], a significant challenge resides in automatically synthesizing approximate circuits (i.e., without manually relying on the expertise of designers). The accuracy of the results in approximate circuits is evaluated based on several error metrics such as worst-case error, bit-flip error, or error-rate, that measure the approximate error by comparing the output of the exact circuit against that of the approximate one. Therefore, it is important to automatically generate approximate circuits that, by design, respect the requirements on the error metrics.

1.2 Project Motivation

A preliminary phase of this project concerns circuit visualization, which is a powerful instrument for getting insights on approximate circuits. In order to depict a circuit graphically, we use the *dot* tool. This takes a *GV* file textually describing a circuit written using the graph description language *DOT* in input and produces a new file containing its corresponding graphical representation. The output of *dot* is described in detail after introducing the *GV* format in section 3.1.2.
Figure 1 shows the graphical representation of a simple *GV* file. The circuit described in it holds a NOT gate negating the value in input on the left branch, and an ASSIGN gate renaming the input on the right. Their values are forwarded to an AND gate, which returns the Boolean value true only if both the values in input are true. The circuit will always return false since the AND gate receives the initial value and its negated value.

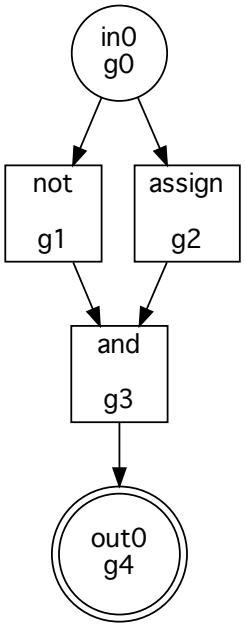

Figure 1. Simple file graphical representation obtained with *dot*

1.3 Project Goals

1.3.1 Implementing a Translator from Blif to GV

A commonly used text format for circuit description is the *Berkeley Logic Interchange Format* (BLIF). This project's first objective consists of designing and implementing a translator that, given a *blif* file describing a circuit, takes that file in input, processes and parses it, and returns the corresponding *GV* file as output. Both the formats are exhaustively explained in sections 3.1.1 and 3.1.2, respectively. Furthermore, the translator should be able to perform other operations after converting the given file, such as removing useless and redundant gates in order to obtain a simpler version of the processed circuit with the same semantical meaning.

1.3.2 Experiments and Analysis with MUS

The second part of this project deals with *Minimal Unsatisfiable Subsets* (MUSes), meaning approximate circuits obtained by removing gates from a given circuit through SAT [5]. We perform experiments with the MUS toolchain, which, given a benchmark in *GV* format, executes a series of operations to create different versions of the given file. Finally, it finds all the possible *MUSes* within a pre-defined timeout value and with respect to a pre-defined error threshold. MUS theory will be illustrated in section 3.2.3.

2 State of the Art

Nowadays, the state of the art provides a variety of works in SAT-based approximate logic synthesis. One of the most recent methods is called the "*MUS-based Circuit Approximation Technique*". (MUSCAT) [6] and generates "valid-by-construction" approximate circuits by inserting cut-points to replace connections between gates by constant values, allowing for subsequent logic minimization. MUSCAT aims to utilize formal verification engines to identify *Minimal Unsatisfiable Subsets* (MUS), determining the maximum number of non-violating cut-points. MUSCAT's solution is optimal with respect to the number of activated cut-points, also providing a guarantee on the quality constraints. The theory behind MUS will be introduced in section 3.2.3.
Another method among AC's most important and diffuse ones is *And-Inverter Graphs* (AIGs) [7]. It is based on rewriting operations that improve performance and ensures the bounds of approximation errors introduced through formal guarantees on error metrics. This synthesis approach is evaluated on various designs to show its usefulness and applicability. The results obtained with AIGs synthesis show comparable quality to manually hand-crafted approximate designs. Furthermore, AIGs are capable of trading off the relative significance of each error metric for a particular application to improve the quality of the synthesized circuits.
The last state-of-the-art model we introduce in this section is the *BMF-based Logic Approximate Synthesis* (BLASYS) [8], which presents a novel paradigm to synthesize approximate circuits using modified algorithms for Boolean matrix factorization (BMF). This methodology approximates the truth table of a sub-circuit through BMF to a controllable approximation degree; the factorization results are then used to synthesize a less complex sub-circuit. BLASYS obtains a smooth trade-off between accuracy and circuit complexity, measured by area and energy consumption.

3 Project Design and Implementation

3.1 Blif2GV Translator

The first part of this report describes the implementation of the *Blif2GV Translator*: an application that offers a clear graphical view of a circuit. This converter inputs a *blif* file holding the textual representation of a Boolean circuit and outputs the corresponding file in GV format. Both the file formats are exhaustively described in sections 3.2.2 and 3.2.3, respectively.
The entire translator was developed using the Python [9] programming language.
The following sections offer a detailed explanation of what happens during each execution phase. Since the Blif2GV Translator deals with Boolean circuits, we must introduce a few notions. *Boolean logic* is the branch of algebra in which the values of the variables are the truth values true and false, usually denoted by the numbers 1 and 0, respectively. George Boole introduced it in the book "*The Mathematical Analysis of Logic*" [1847]. A Boolean operator is a function that takes binary variables in input, processes them, and returns a single binary output (i.e., either 1 or 0). Let us analyze the following example, representing an AND operator:

$$1 \wedge 0 = 0$$

Such operations are defined by *truth tables*, wherein the combinations of inputs and the corresponding output are shown. Given the fact that we could encounter any type of *unary* (i.e., one input - one output) and *binary* (i.e., two inputs - one output) operators while parsing a *blif* file, we should define each possible input(s)-output relation in order to be able to understand each component of a Boolean circuit. Every Boolean operator is exhaustively defined in Appendix A, along with the corresponding truth table.

3.1.1 Introducing the blif Format

The *Berkeley Logic Interchange Format* (BLIF) [10] represents a logic-level hierarchical circuit in textual form. The term *circuit* refers to a combinational and sequential network of Boolean functions which can be viewed as a directed graph, where each node can be broken up into the following components:

- **Input(s)**

  The set of inputs received by the node; it could consist of a single input, in the case of a unary operator, an ordered pair of inputs, in the case of a binary operator, or a sequence of inputs with no pre-defined limits, in the case of a special operator. In the latter is the case, then the node must be declared in a `.subckt`.

- **Operator**

  The operator that processes the input received by the node and returns the corresponding output; it could be a unary operator, such as NOT, a binary operator, such as AND, or a special operator, such as MUX.

- **Output**

  The single output returned by the operator after the latter finishes processing the received input.

The file's body is composed of "commands", which are lines that declare logic gates. The circuit nodes are defined by specifying the input(s), the operator, and the output. There exist two different ways of writing such definitions:

- `.subckt`

  This kind of definition has the following syntax:

  ```
  .subckt $<operator> A=<input> [B=<input2> ...]  Y=<output>
  ```

  Every pair `letter=<name>` but the last one represents an input of the defined node. The following example shows a logical AND gate definition:

  ```
  .subckt $and g0 g1 g2
  ```

- `.names`

  This kind of definition has the following syntax:

  ```
  .names <input> [<input2> ...]  <output>
  <truth-table>
  ```

  The operator is defined by the truth table declared in the lines immediately below. A truth table is defined by one or more lines of numbers, each indicating a boolean relation that returns the logic value true. There must be a sequence of non-separated numbers, one for each input, and a number for the output, white-space-separated by the first sequence. The following example shows a logical AND gate definition:

  ```
  .names g0 g1 g2
  11 1
  ```

The following code snippet represents a simple *blif* file.

```
1  .model circuit
2  .inputs g0
3  .outputs g8
4  .names $false
5  .names $true
6  1
7  .names $undef
8  .subckt $assign A=g0 Y=g1
9  .subckt $assign A=g0 Y=g2
10 .subckt $not A=g1 Y=g3
11 .subckt $not A=g3 Y=g4
12 .subckt $not A=g2 Y=g5
13 .subckt $and A=g4 B=g5 Y=g6
14 .subckt $assign A=g6 Y=g7
15 .subckt $assign A=g7 Y=g8
16 .end
```

3.1.2 Introducing the GV Format

A *Graphviz Dot* (*GV*) file [11] provides the characteristics of a graph and is written using the DOT language. The body of a *GV* file can be broken up into two parts: the nodes declaration and the edges assignment. Each node-declaring line defines a single gate using the following syntax:

```
<node_name> [label="<node_label>" <style>]
```

The name displayed after converting a *GV* file into a pdf is the value of the property `label`. Each edge-declaring line defines a link from a source node to a destination node with the following syntax:

```
<source_node> -> <destination_node>
```

The following code snippet represents the *GV* file corresponding to the *blif* file defined at the end of section 3.1.1.

```
digraph circuit {
    node[style=filled, fillcolor=white, shape=rect, fontname=geneva]
    g0 [label="in0\ng0", shape=circle, fillcolor=white]
    g1 [label="assign\n\ng1", fillcolor=white]
    g2 [label="assign\n\ng2", fillcolor=white]
    g3 [label="not\n\ng3", fillcolor=white]
    g4 [label="not\n\ng4", fillcolor=white]
    g5 [label="not\n\ng5", fillcolor=white]
    g6 [label="and\n\ng6", fillcolor=white]
    g7 [label="assign\n\ng7", fillcolor=white]
    g8 [label="assign\n\ng8", fillcolor=white]
    g8 [label="out0\ng8", shape=doublecircle, fillcolor=white]
    edge [fontname=Geneva, fontcolor=forestgreen]
    g0 -> g1
    g0 -> g2
    g1 -> g3
    g3 -> g4
    g2 -> g5
    g4 -> g6
    g5 -> g6
    g6 -> g7
    g7 -> g8
}
```

Figure 2 shows the graphical representation of this file. The described circuit is more complex than the one seen in section 2, but has the same semantical meaning: gate 6 is an AND gate that takes in input the result of two consecutive NOT gates from the left branch and the value of a single NOT gate from the right one; thus, the AND gate will always evaluate to false.

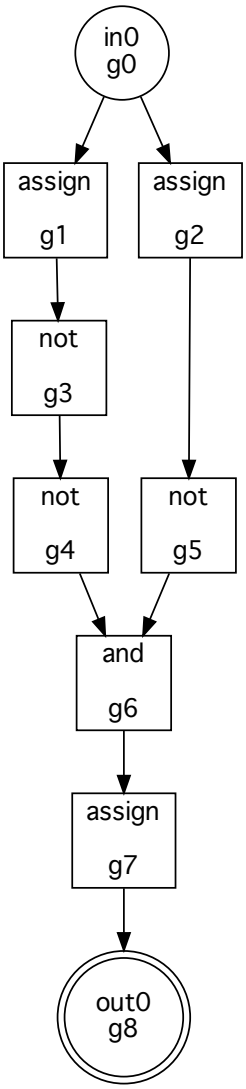

Figure 2. File graphical representation obtained with *dot*

3.1.3 Blif2GV Algorithm Description

Since we deal with logic circuits, we need to implement a way of representing a circuit. A circuit object has a set of inputs, a set of outputs, and a set of sub-circuits, and it is created by parsing a *blif* file's header. Moreover, we must design a way of defining the objects a circuit is made of: the sub-circuits. A sub-circuit object has a set of inputs, an operator, and a single output and represents a circuit gate.
Algorithm 1 shows the pseudo-code of the parsing method performed by the *Blif2GV Translator*.

Algorithm 1 `blif_parser(file_name)`

```
generate ckt object from file_name
set subckts to []
open file_name
for each line in file_name.lines do
    create subckt
    append subckt to subckts
end for
close file_name
assign subckts to ckt
assign children and parents to each subckt
simplify ckt
return ckt
```

In order to parse `.name` lines, we must perform a few more operations.
In the case of such a line, we need to parse the first line to get the input(s) and the output; then, we must parse the truth table defined in the following lines to assign the corresponding operator. The encounterable truth tables are defined in dictionaries created by permuting each operator definition. As a result, a truth table is identified independently of how it is declared.

3.1.4 Simplifying a Circuit

After implementing the translator, we wish to program more features to generate more simple circuits.

Fixing the Syntax

Given that the syntax of a *GV* file rejects several symbols that, in contrast, could be used in a *blif* file, we must modify everything that could lead to an error. More specifically, the *GV* syntax does not accept non-alphanumeric characters at the beginning of a gate name and many special symbols inside it.
Let us examine the following example, taken from *BLASYS*:

```
.subckt $and A=$not$abs_.v:164$307_Y B=$not$abs_.v:164$308_Y
                    Y=$and$abs_.v:164$309_Y
                                ⇓
    1) input: n164_307_Y  operator: and  output: n164_309_Y
    2) input: n164_308_Y  operator: and  output: n164_309_Y
```

Removing Useless ASSIGN Gates

The role of an ASSIGN gate is renaming the received input and forwarding it to another gate. Since these gates are introduced by *blif* files and do not alter the semantical meaning of the circuit, we decided to remove such gates. We must ensure that each link is set correctly after the removal in order not to get errors. **Figure 3** shows the graphical representation of the circuit defined in **Figure 2** after performing the first removal operation: all the ASSIGN gates are removed.

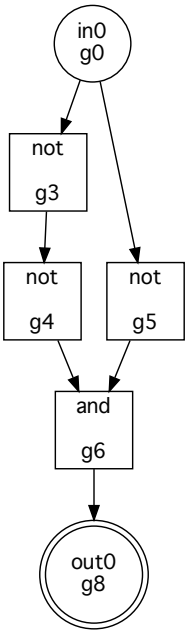

Figure 3. ASSIGN gates removal result

Removing Redundant NOT Gates

A NOT gate changes the value of the input received into its complement (i.e., from true to false and vice-versa). Two or more NOT gates are redundant if the value outputted by the last gate of the sequence is equal to the value taken in input by the first gate.
Figure 4 shows the graphical representation of the circuit defined in **Figure 2** after performing the second removal operation: the redundant NOT gates are entirely removed.

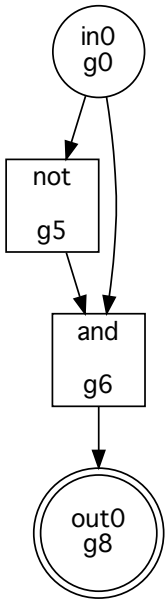

Figure 4. NOT gates removal result

3.1.5 Executing the Translator

There are two different manners for executing the translator: a *Text-based User Interface* (TUI) or a *Graphical User Interface* (GUI).

Text-based User Interface

In order to run the translator with the TUI, it is sufficient to type the following command while being in the project's root directory:

```
python3 main.py <blif_file_name> [dot]
```

The placeholder `<blif_file_name>` must be replaced by the path of the *blif* file that the user wants to convert. Moreover, it is possible to add the argument `dot` to convert the newly generated *GV* file into a pdf using DOT at the end of the translation process.
Depending on the result of the translation process, a different message will be displayed in the standard output. The code handles several types of errors.

Graphical User Interface

In order to run the translator with the GUI, it is sufficient to execute the python script `main.py` with no additional arguments while being in the project's root directory:

```
python3 main.py
```

The script will then display a GUI with the following initial window: **Figure 5** shows the initial window of the GUI.

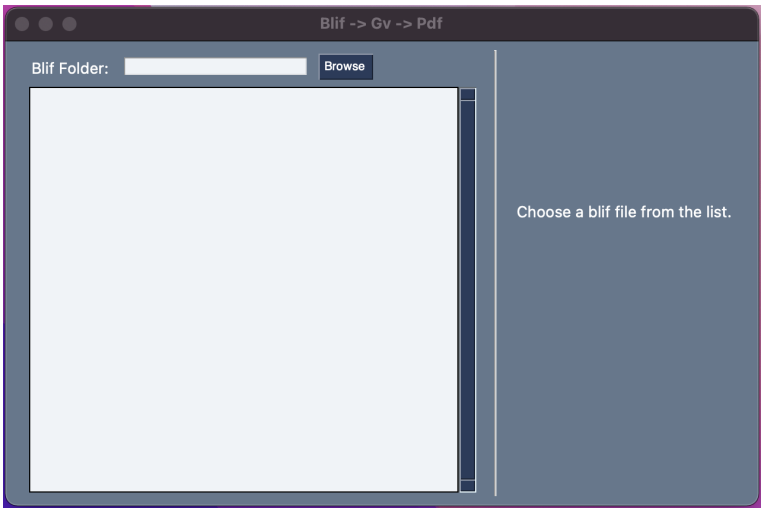

Figure 5. GUI - initial window

In the top-center part of the window, there are a text area and a button labeled *Browse*: the user can either type the path to a folder containing at least one *blif* file or search for it by clicking on the button. If the folder selected by the user contains at least one *blif* file, then the window will display a list of all the *blif* files found in that directory on the left. The window will display an error if the selected folder holds no *blif* files.
By clicking on one of the displayed *blif* files, the corresponding line of the list will be highlighted. The window will then display the name of the *blif* file on the left, along with a button labeled *Convert*: if the user clicks on the latter, then the selected *blif* file will be converted to *GV* and pdf.

Figure 6 shows the window that appears when the input *blif* file is successfully converted to *GV*, and if the corresponding pdf file is succesfully generated by DOT.

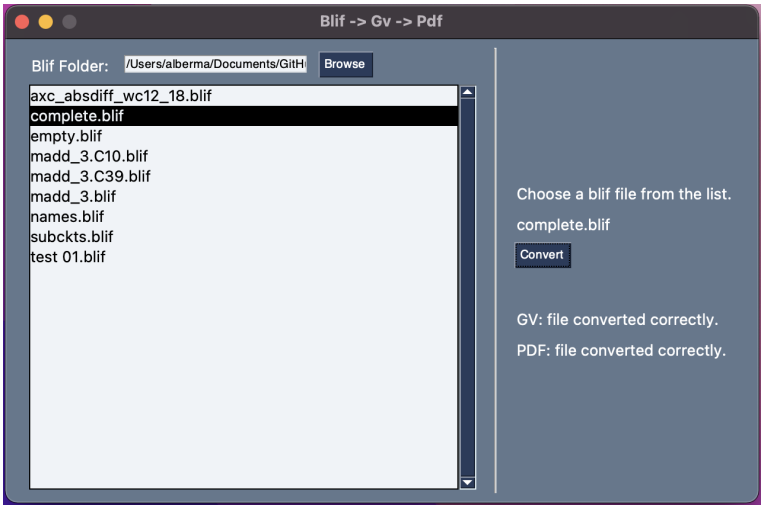

Figure 6. GUI - successful conversions window

Finally, depending on the result of the translation process, a different message will be displayed on the window. The code handles several types of errors.

3.2 SAT for Approximate Synthesis

3.2.1 Introduction to SAT Theory

The *Boolean satisfiability problem* (SAT) considers a formula containing binary variables connected by logical relations such as OR and AND. It aims to establish whether there is any way to set these variables so that the formula evaluates to true.
The *Boolean satisfiability* problem asks whether there exists at least one combination of input variables $x_i \in \{\text{true}, \text{false}\}$ for which a Boolean formula returns true, i.e. it is *satisfiable*.
A *SAT solver* [5] is an algorithm for establishing satisfiability: it takes a Boolean formula in input and returns SAT if it finds a combination of variables that satisfies it; otherwise, it returns UNSAT. It may return without an answer if it cannot determine whether the problem is SAT or UNSAT.
SAT solvers can be classified into two types: *complete algorithms*, which guarantee to return either SAT or UNSAT, and *incomplete algorithms*, which could return UNKNOWN.
The first step for solving a SAT problem is converting the formula in input to a standard form known as *conjunctive normal form*, which consists of a conjunction of disjunctions and is more amenable to algorithmic manipulation:

$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

Each term is called a *clause* and combines variables and their complements with OR operators. Clauses are combined via AND operators. In a *conjunctive normal form*, a variable is known as a *literal*. We can characterize a problem based on the number and size of the clauses in the conjunctive normal form: given that a *k-clause* contains k variables, then a problem is *k-SAT* if all the clauses contain k variables.
A solver which finds a solution returns a *certificate* along with it. A certificate is used to check the result with a simpler algorithm: if the solver returned SAT, then the certificate will be a set of variables satisfying the formula; otherwise, the certificate will be a complex data structure depending on the solver.

In the *UNSAT problem*, we aim to show that no combination of literals satisfies a given formula: *UNSAT solvers* return as soon as they establish that the formula is UNSAT, but it may take exponential time. In general, SAT-problems are NP-hard.

3.2.2 Minimal Unsatisfiable Subsets

Given an unsatisfiable Boolean formula ϕ, an unsatisfiable subset of clauses U of ϕ is called *Minimal Unsatisfiable Subset* (*MUS*) if every proper subset of U is satisfiable. Moreover, a minimum unsatisfiable subset contains the smallest number of the original clauses required to be still unsatisfiable. The *MUSes* of a Boolean formula serve as explanations for the unsatisfiability of the formula itself. The availability of efficient SAT solvers has aided the development of techniques for finding and enumerating *MUSes*.
A generic combinatorial circuit can always be expressed as a Boolean formula. This formula can be manipulated by a SAT solver to obtain an approximate one that slightly deviates from the original. Indeed, with the proper problem formulation, *MUSes* returned by the solver can be interpreted as approximate circuits.
The second part of this project deals with finding and enumerating the *MUSes* of a given benchmark through a toolchain that uses a modified version of *MUST* [12], then analyzing the area of the smallest MUS found with respect to an error threshold, since designers are interested in the smallest circuit with equal functionality.

4 MUST Results Analysis

4.1 Experimental Settings

Given a set of benchmarks, we must find the maximum number of *MUSes* for each among them, along with their areas. More precisely, we aim at obtaining the MUS representing the smallest approximate circuit with respect to an error threshold (ET) and a timeout value. We decided to set the timeout value to 30 to limit the execution time. After running the toolchain on a benchmark, we use *yosys* [13] to get the minimal area among the newly-created *MUSes*. Since the used *yosys* script requires a *verilog* file [14], we must use the *abc* tool [15] to convert the blif version of the considered benchmark to the *verilog* format; the command employed is the following:

```
abc -c "read_blif <benchmark>.blif ; write_verilog <benchmark>.v"
```

Finally, given the number of *MUSes* found, the script `measure_area_yosys.py` returns each *MUS*' area value along with the minimal one. The command to use is the following:

```
python3 measure_area_yosys.py <benchmark>.v <# MUSes>
```

4.2 Summary of the main Results

This section aims at explaining the results obtained while testing the *MUS* toolchain for a set of benchmarks. The first line of each point in the list declares the name of the corresponding benchmark and reports the number of input constraints along with the value of the area of the exact circuit. The error thresholds used while computing the MUSes areas are 1/8, 2/8, 3/8, 4/8, 5/8, and 6/8 of the maximal error possible for the given circuit, which can be evaluated using the following formula:

$$\text{err} = 2^o - 1,$$

where o is the number of bits in output. The results for each benchmark are shown in a table, which contains the area of the smallest MUS found with respect to the different error thresholds defined in the previous paragraph. Then, the area of every MUS found is represented by a point in a graph. The experiments were run on a set of different arithmetic benchmarks of various bitwidth. **Table 1** summarizes the characteristics of the benchmarks used.

Name	Acronym	Primary In/Out	Size (# Logic Gates)
absolute_difference_9bit	abs_diff_9	18/9	220.57
absolute_difference_10bit	abs_diff_10	20/10	194.76
adder_4bit	adder_4	8/5	46
adder_5bit	adder_5_0.3	10/6	57
adder_6bit	adder_6_0.3	12/7	79
adder_8bit	adder_8_0.5	16/9	99
multiply_add_3bit	madd_3	9/6	105
multiplier_6bit	mul_6	12/12	355
online_adder_4bit	online_adder_4	18/10	119
online_adder_8bit	online_adder_8	34/18	107
sad_4bit	sad_4	20/5	347

Table 1. List of used benchmarks

The following list shows the results obtained for each benchmark:

- **`absolute_difference_10bit`**
  The benchmark `absolute_difference_10bit` has an exact area of 220.57, a number of constraints of 187, and the maximum value for error is 1023. **Table 2** and **Figure 7** display the obtained results for `absolute_difference_10bit`.

Error Threshold	# MUSes	Minimal Area
127	1	147.36
255	1	130.47
383	5	126.24
511	167	18.3
639	143	15.49
767	112	9.39

Table 2. `abs_diff_10` smallest MUSes areas

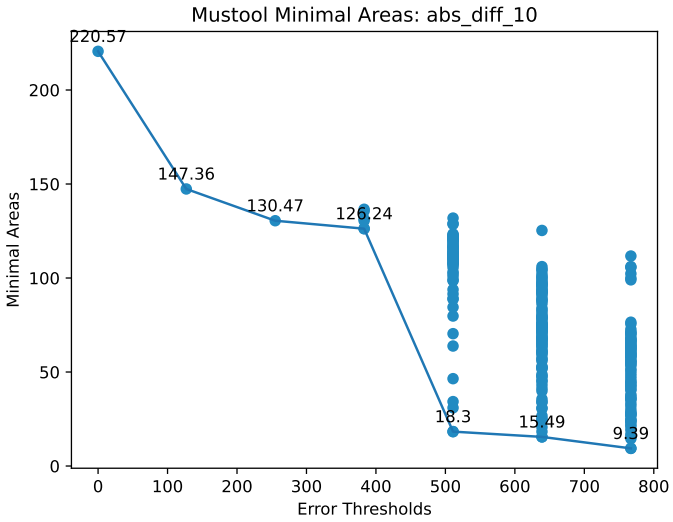

Figure 7. `abs_diff_10` MUSes areas

- **`absolute_difference_9bit`**
  The benchmark `absolute_difference_9bit` has an exact area of 194.76, a number of constraints of 180, and the maximum value for error is 511. **Table 3** and **Figure 8** display the obtained results for `absolute_difference_9bit`.

Error Threshold	# MUSes	Minimal Area
63	1	136.57
127	1	100.43
191	5	100.43
255	167	18.3
319	143	18.3
383	112	9.39

Table 3. `abs_diff_9` smallest MUSes areas

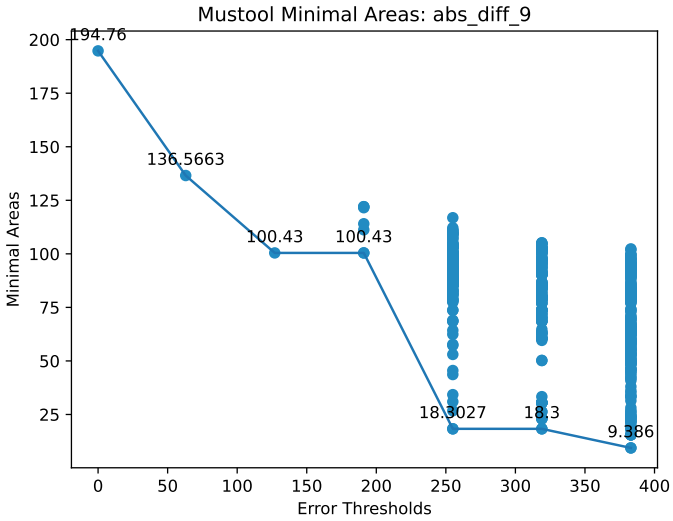

Figure 8. `abs_diff_9` MUSes areas

- **`adder_4bit`**
  The benchmark `adder_4bit` has an exact area of 54.44, a number of constraints of 46, and the maximum value for error is 31. **Table 4** and **Figure 9** display the obtained results for `adder_4bit`.

Error Threshold	# MUSes	Minimal Area
3	2	38
7	4	23
11	11	19.24
15	6	7.51
19	22	7.51
23	15	2.82

Table 4. `adder_4` smallest MUSes areas

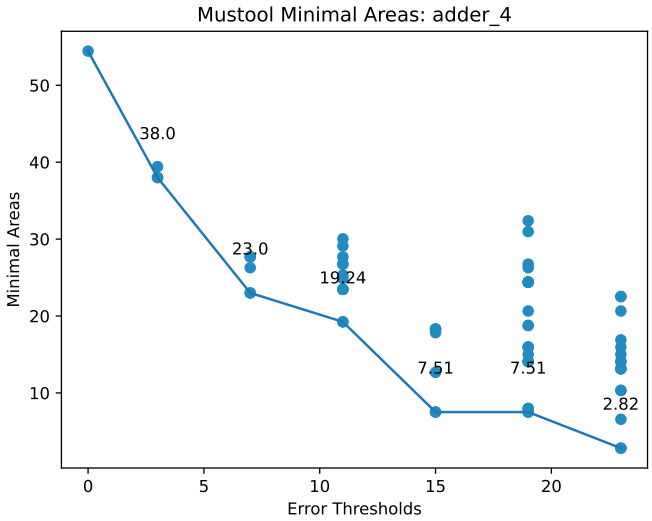

Figure 9. `adder_4` MUSes areas

- **adder_5bit**
  The benchmark adder_5bit has an exact area of 74.62, a number of constraints of 57, and the maximum value for error is 63. **Table 5** and **Figure 10** display the obtained results for adder_5bit.

Error Threshold	# MUSes	Minimal Area
7	14	34.73
15	52	19.24
23	97	9.39
31	124	5.63
39	231	5.63
47	221	2.82

Table 5. adder_5_0.3 smallest MUSes areas

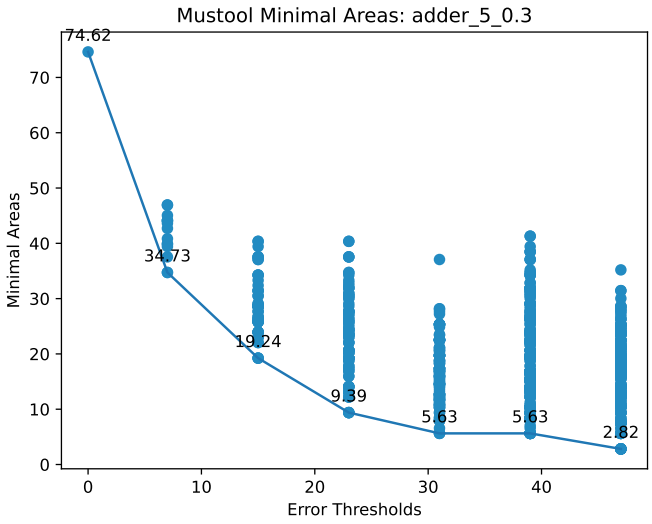

Figure 10. adder_5_0.3 MUSes areas

- **adder_6bit**
  The benchmark adder_6bit has an exact area of 84.47, a number of constraints of 79, and the maximum value for error is 127. **Table 6** and **Figure 11** display the obtained results for adder_6bit.

Error Threshold	# MUSes	Minimal Area
15	75	31.91
31	138	22.06
47	138	14.08
63	223	2.82
79	376	2.82
95	136	2.82

Table 6. adder_6_0.3 smallest MUSes areas

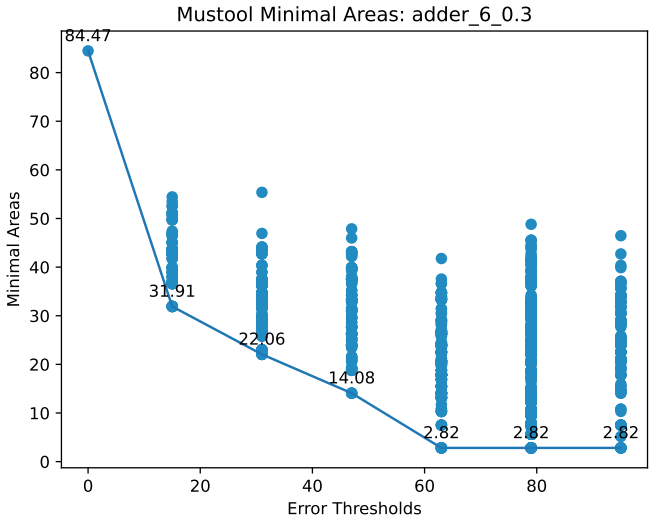

Figure 11. `adder_6_0.3` MUSes areas

- **`adder_8bit`**
  The benchmark `adder_8bit` has an exact area of 115.92, a number of constraints of 99, and the maximum value for error is 511. **Table 7** and **Figure 12** display the obtained results for `adder_8bit`.

Error Threshold	# MUSes	Minimal Area
63	20	39.42
127	24	22.53
191	81	9.39
255	28	7.51
319	81	5.63
383	63	2.82

Table 7. `adder_8_0.5` smallest MUSes areas

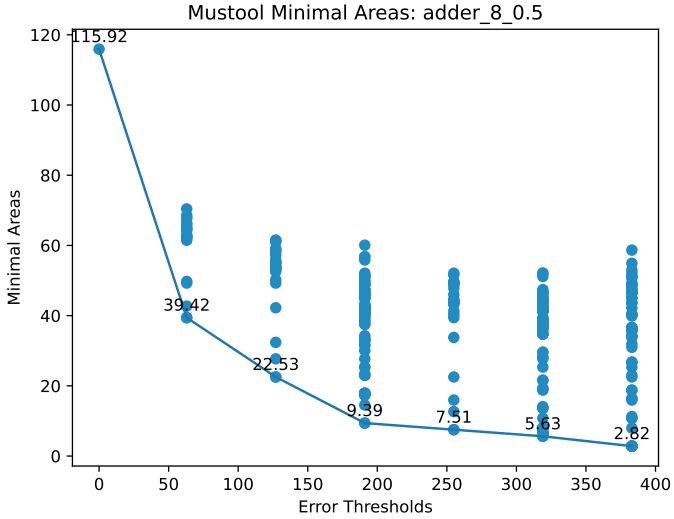

Figure 12. `adder_8_0.5` MUSes areas

- **`multiply_add_3bit`**
  The benchmark `multiply_add_3bit` has an exact area of 127.65, a number of constraints of 105, and the maximum value for error is 63. **Table 8** and **Figure 13** display the obtained results for `multiply_add_3bit`.

Error Threshold	# MUSes	Minimal Area
7	165	56.79
15	200	28.63
23	220	8.45
31	306	8.45
39	314	8.45
47	215	2.82

Table 8. `madd_3` smallest MUSes areas

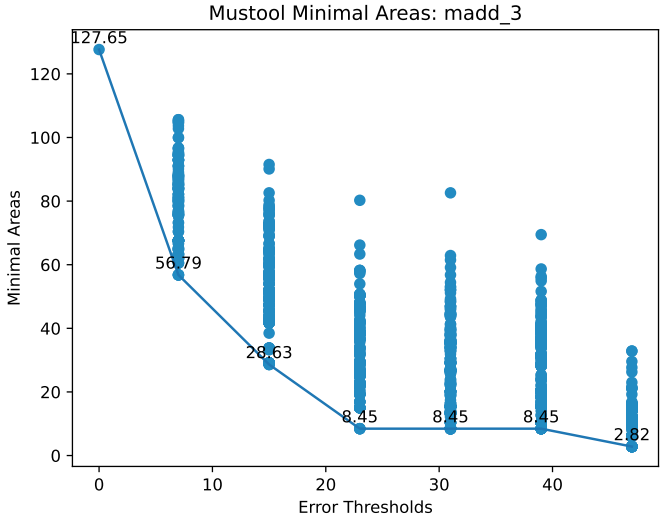

Figure 13. `madd_3` MUSes areas

- **`multiplier_6bit`**
  The benchmark `multiplier_6bit` has an exact area of 481.5, a number of constraints of 355, and the maximum value for error is 4095. **Table 9** and **Figure 14** display the obtained results for `multiplier_6bit`.

Error Threshold	# MUSes	Minimal Area
511	15	152.52
1023	62	63.36
1535	63	22.53
2047	143	8.45
2559	78	8.45
3071	221	2.82

Table 9. `mul_6` smallest MUSes areas

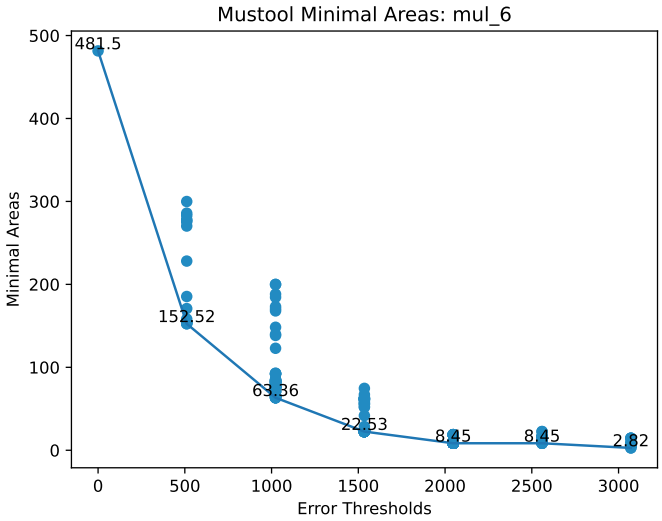

Figure 14. `mul_6` MUSes areas

- **`online_adder_4bit`**
  The benchmark `online_adder_4bit` has an exact area of 127.65, a number of constraints of 119, and the maximum value for error is 1023. **Table 10** and **Figure 15** display the obtained results for `online_adder_4bit`.

Error Threshold	# MUSes	Minimal Area
127	1	54.91
255	1	30.5
383	17	28.63
511	1	7.04
639	2	7.04
767	2	7.04

Table 10. `online_adder_4` smallest MUSes areas

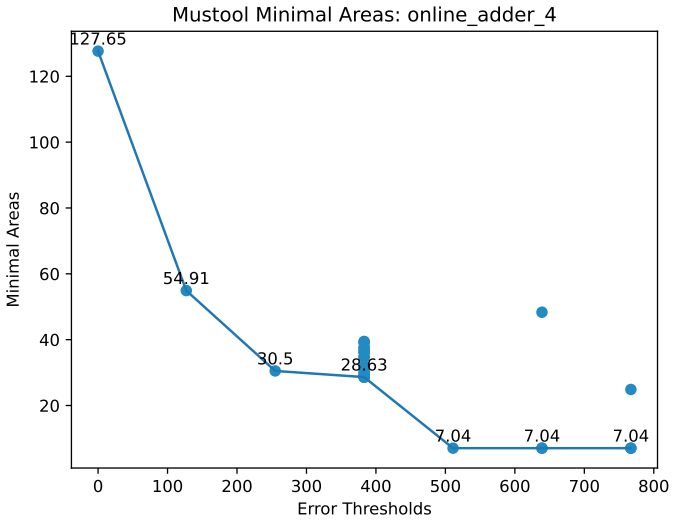

Figure 15. `online_adder_4` MUSes areas

- **`online_adder_8bit`**
  The benchmark `online_adder_8bit` has an exact area of 254.83, a number of constraints of 107, and the maximum value for error is 262143. **Table 11** and **Figure 16** display the obtained results for `online_adder_8bit`.

Error Threshold	# MUSes	Minimal Area
32767	1	55.85
65535	1	30.5
98303	11	28.63
131071	1	7.04
163839	2	7.04
196607	2	7.04

Table 11. `online_adder_8` smallest MUSes areas

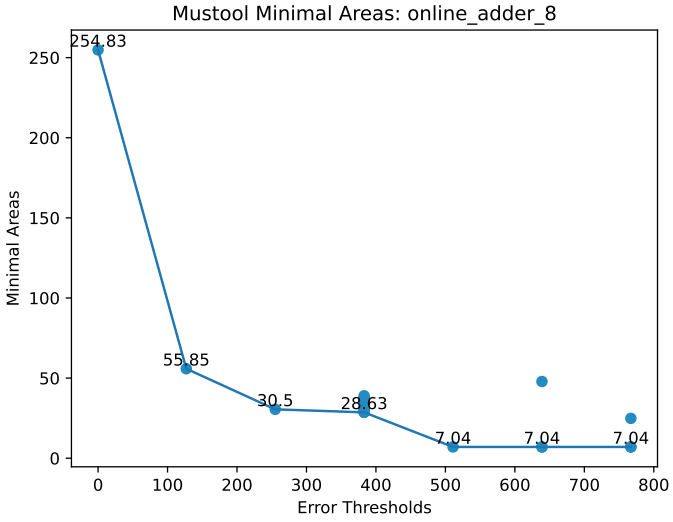

Figure 16. `online_adder_8` MUSes areas

- **`sad_4bit`**
  The benchmark `sad_4bit` has an exact area of 438.8, a number of constraints of 347, and the maximum value for error is 31. **Table 12** and **Figure 17** display the obtained results for `sad_4bit`.

Error Threshold	# MUSes	Minimal Area
3	1	421.43
7	1	409.7
11	2	406.41
15	1	402.66
19	2	389.99
23	2	380.13

Table 12. `sad_4` smallest MUSes areas

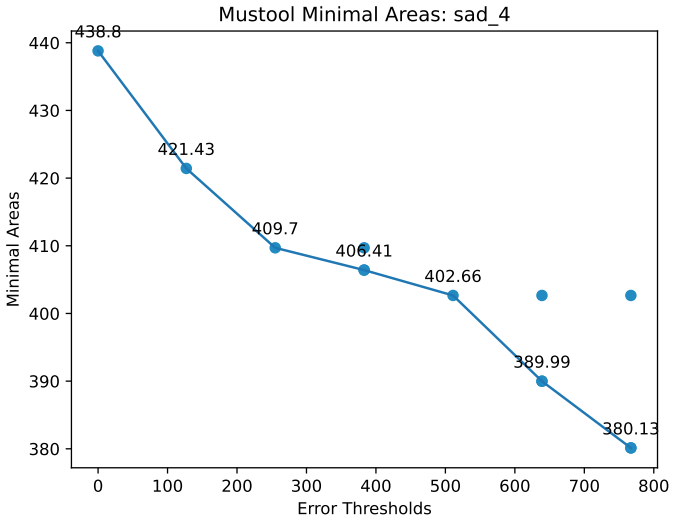

Figure 17. `sad_4` MUSes areas

In order to better understand the differences between an exact circuit and one of its approximate versions, let us analyze a comparison between them, with an error threshold of 10.
Figure 18 and **Figure 19** show the exact circuit and an approximate one of benchmark `adder_4`, respectively. We can observe that many gates have been set to zero in the generated approximate circuit.

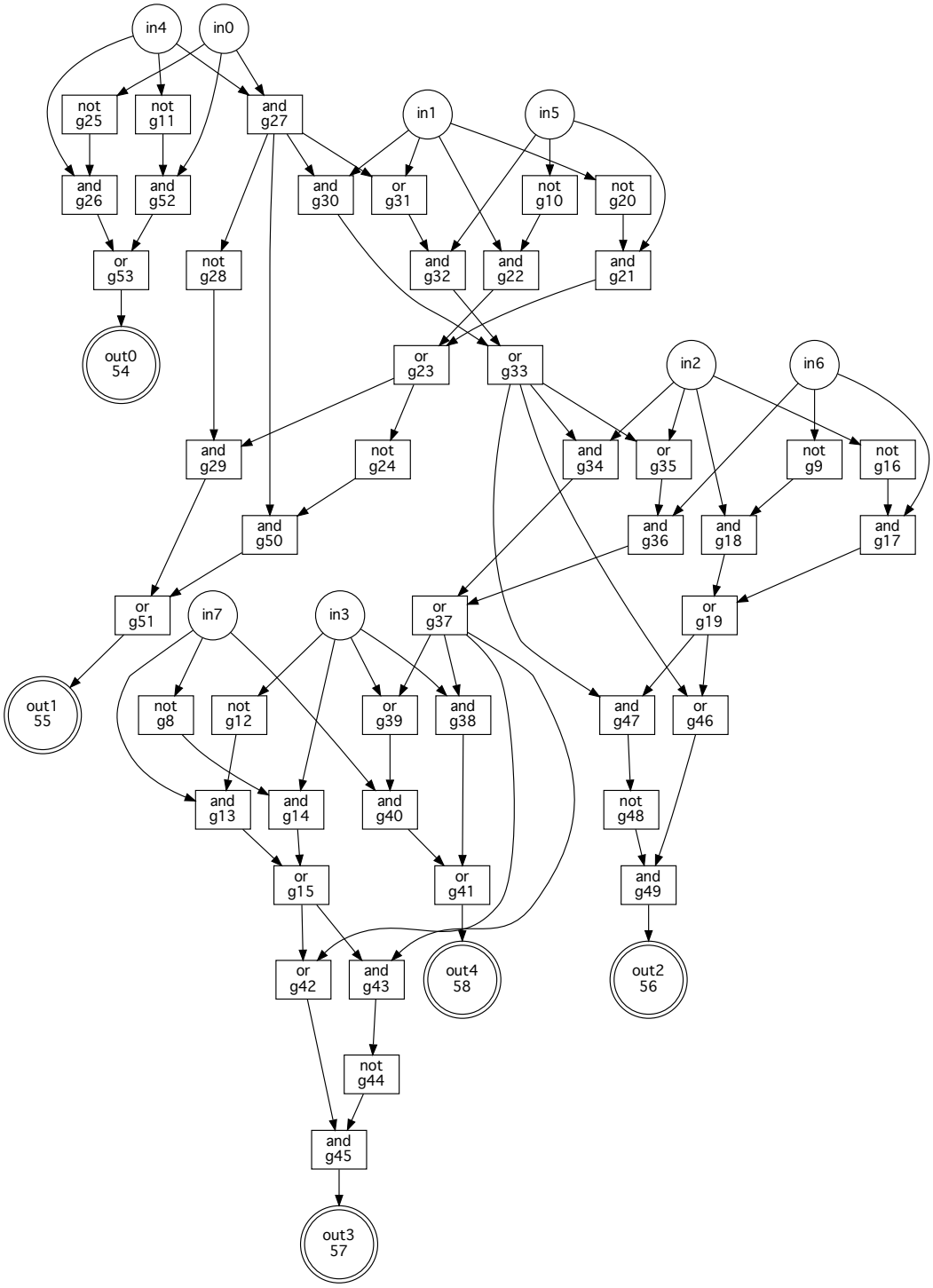

Figure 18. `adder_4` exact circuit graphical representation

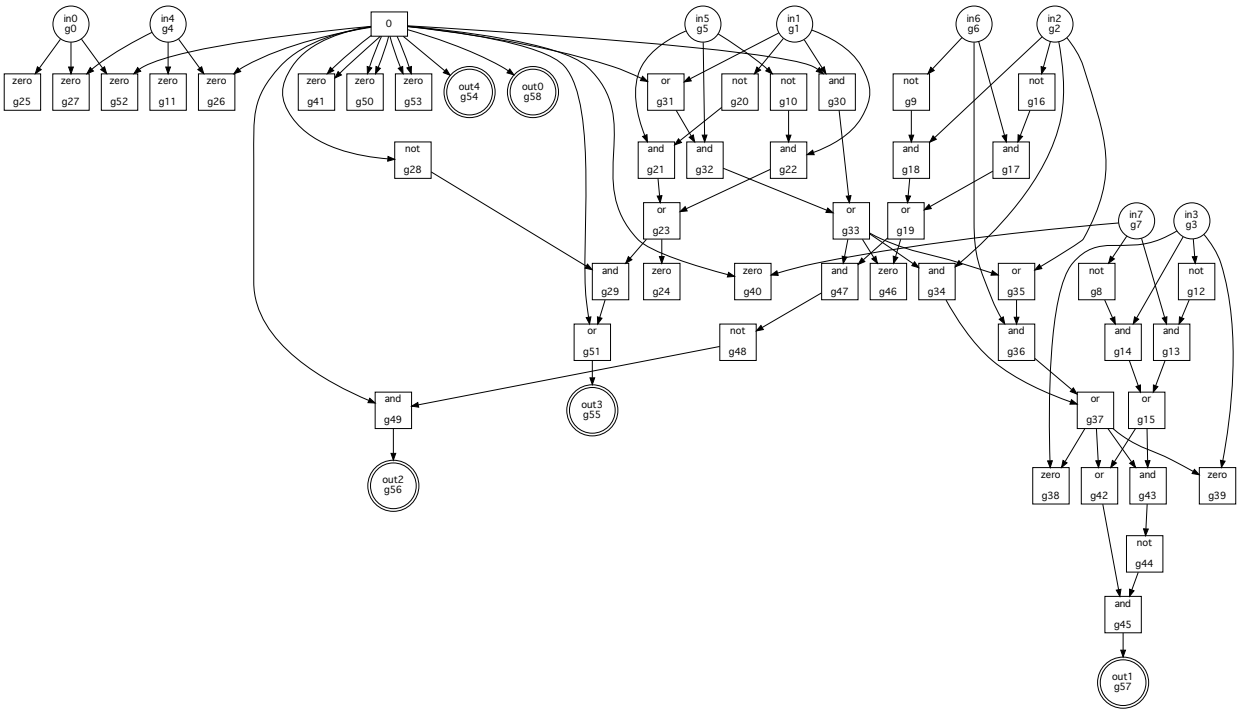

Figure 19. `adder_4` approximate circuit graphical representation (ET = 10)

5 Conclusion and Future Work

5.1 Concluding Remarks

In the first part of this project, we implemented a translator capable of converting *blif* files into their corresponding version in the *GV* format, while also simplifying them. Finally, the application generates a pdf file with dot, graphically rendering the circuit of the input file. The code and documentation of the *Blif2GV Translator* can be found in its Github repository [16].
In the second part of this project, we performed various experiment on a set of different benchmarks by using *MUST*, along with some constraints. We observed that *MUST* is capable of rapidly synthesizing small approximate circuits. This proves the efficiency of SAT-based approximate synthesis, and leaves room for further exploration of the topic.
The *Blif2GV Translator* was also helpful in this second part since, after generating a series of *blif* files representing the approximate circuits of the input file, we could use it in order to display the exact circuit and all the approximate ones, getting a clear graphical view of the differences between them as shown in **Figure 18** and **Figure 19**.

5.2 Future Work

One possible future enhancement of the *Blif2GV Translator* could be integrating it into other tools for approximate logic synthesis. For example, we could develop an "*approximate circuit visualization*" tool that takes approximate circuits written in the *verilog* format, transforms them into their *blif* counterparts with the *abc* tool, and finally converts the newly-obtained files into *GV* through the *Blif2GV Translator*.
For what concerns the second part of this project (i.e., analysis on MUSes), after having quantified the approximate circuits' areas, we could compare the obtained results with those returned by other methods of the state of the art in order to assert on MUST effectiveness. Moreover, we could think of possible heuristics to improve MUST scalability for large circuits (e.g., 64-bit adders). Indeed, MUST is currently incapable to deal with such circuits and, hence, the relaxation of optimality requirements could represent a valid strategy.

6 Appendix A - Boolean Operators

6.1 Definition of Boolean Unary Operators

A *Boolean unary operation* consists of a single input, a *unary operator*, and a single output. In order to better understand what a Boolean unary operator looks like, let us examine the following example:

$$\Phi = (\neg x_1)$$

The Boolean operator ¬ is known as "NOT" and, in this case, takes in input variable x_1, processes it, and returns a single output.
The following list explains the working of all the logical unary operators:

- **ZERO** (unary)

  The unary ZERO operator takes an input x and returns false, regardless of the value of x:

  $$\text{ZERO}_1(x) = 0$$

  The subscript "$_1$" is needed to distinguish between the unary ZERO operator and the binary one.
  Table 13 shows the truth table of the unary ZERO operator.

Input	Output
0	0
1	0

Table 13. Unary ZERO operator truth table

- **ASSIGN**

  The ASSIGN operator takes an input x and returns true if the value of x is true or false if the value of x is false:

  $$\text{ASSIGN}(x) = x$$

  It is typically used to rename the input without changing its value.
  Table 14 shows the truth table of the ASSIGN operator.

Input	Output
0	0
1	1

Table 14. ASSIGN operator truth table

- **NOT**

  The NOT operator takes an input x and returns the complement of x; thus, it returns true if the value of x is false or false if the value of x is true:

  $$\neg x = \overline{x}$$

It is equivalent to the negated form of the ASSIGN operator:

$$\neg x \iff \neg(\text{ASSIGN}(x))$$

Table 15 shows the truth table of the NOT operator.

Input	Output
0	1
1	0

Table 15. NOT operator truth table

- **ONE**

  The unary ONE operator takes an input x and returns true, regardless of the value of x:

  $$\text{ONE}_1(x) = 1$$

  The subscript "$_1$" is needed to distinguish between the unary ONE operator and the binary one.
  It is equivalent to the negated form of the unary ZERO operator:

  $$\text{ONE}_1(x) \iff \neg(\text{ZERO}_1(x))$$

  Table 16 shows the truth table of the unary ONE operator.

Input	Output
0	1
1	1

Table 16. Unary ONE operator truth table

6.2 Definition of Boolean Binary Operators

A *Boolean binary operation* consists of two inputs, a *binary operator*, and a single output. In order to better understand what a Boolean binary operator looks like, let us examine the following example:

$$\Phi = (x_1 \wedge x_2)$$

The Boolean operator $\wedge$ is known as "AND" and, in this case, takes in input variables x_1 and x_2, processes them, and returns a single output.
The following list explains the working of all the logical binary operators:

- **ZERO** (binary)

  The binary ZERO operator takes two inputs, x and y, and returns false, regardless of the values of both x and y:

  $$\text{ZERO}_2(x, y) = 0$$

The subscript "$_2$" is needed to distinguish between the binary ZERO operator and the unary one.
Table 17 shows the truth table of the binary ZERO operator.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	0

Table 17. Binary ZERO operator truth table

- **AND**

  The AND operator takes two inputs, x and y, and returns true only if the values of x and y are both true; otherwise, it returns false:

  $$x \wedge y$$

  Table 18 shows the truth table of the AND operator.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Table 18. AND operator truth table

- **NOT_IMPLY**

  The NOT_IMPLY operator takes two inputs, x and y, and returns true only if the values of x and y are true and false, respectively; otherwise, it returns false:

  $$x \overline{\Longrightarrow} y$$

  It is equivalent to the negated form of the IMPLY operator:

  $$x \overline{\Longrightarrow} y \iff \neg(x \Longrightarrow y)$$

  Table 19 shows the truth table of the NOT_IMPLY operator.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	1
1	1	0

Table 19. NOT_IMPLY operator truth table

- **ONE_DC**

  The ONE_DC operator takes two inputs, x and y, and returns true only if the value of the first input (i.e., x) is true, regardless of the value of the second input (i.e., y); otherwise, it returns false:

$$\text{ONE\_DC}(x, y) = x$$

  Table 20 shows the truth table of the ONE_DC operator.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	1
1	1	1

Table 20. ONE_DC operator truth table

- **ZERO_ONE**

  The ZERO_ONE operator takes two inputs, x and y, and returns true only if the values of x and y are false and true, respectively; otherwise, it returns false:

$$\text{ZERO\_ONE}(x, y) = 1$$

  Table 21 shows the truth table of the ZERO_ONE operator.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	0
1	1	0

Table 21. ZERO_ONE operator truth table

- **DC_ONE**

  The DC_ONE operator takes two inputs, x and y, and returns true only if the value of the second input (i.e., y) is true, regardless of the value of the first input (i.e., x); otherwise, it returns false:

$$\text{DC\_ONE}(x, y) = 1$$

  Table 22 shows the truth table of the DC_ONE operator.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	0
1	1	1

Table 22. DC_ONE operator truth table

- **XOR**

  The XOR operator takes two inputs, x and y, and returns true only if exactly one of the values of x and y is true; otherwise, it returns false:

  $$x \oplus y$$

  Table 23 shows the truth table of the XOR operator.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 23. XOR operator truth table

- **OR**

  The OR operator takes two inputs, x and y, and returns true only if at least one of the values of x and y is true; otherwise, it returns false:

  $$x \vee y$$

  Table 24 shows the truth table of the OR operator.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table 24. OR operator truth table

- **NOR**

  The NOR operator takes two inputs, x and y, and returns true only if both the values of x and y are false; otherwise, it returns false:

  $$x \overline{\vee} y$$

  It is equivalent to the negated form of the OR operator:

  $$x \overline{\vee} y \iff \neg(x \vee y)$$

  Table 25 shows the truth table of the NOR operator.

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	0

Table 25. NOR operator truth table

- **EQUALITY**

  The EQUALITY operator takes two inputs, x and y, and returns true only if the values of x and y are equal (i.e., they are both true or false); otherwise, it returns false:

  $$x \iff y$$

  It is equivalent to the negated form of the XOR operator:

  $$(x \iff y) \iff \neg(x \oplus y)$$

  Table 26 shows the truth table of the EQUALITY operator.

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	1

Table 26. EQUALITY operator truth table

- **DC_ZERO**

  The DC_ZERO operator takes two inputs, x and y, and returns true only if the value of the second input (i.e., y) is false, regardless of the value of the first input (i.e., x); otherwise, it returns false:

  $$\text{DC\_ZERO}(x, y) = 1$$

  It is equivalent to the negated form of the DC_ONE operator:

  $$\text{DC\_ZERO}(x, y) \iff \neg(\text{DC\_ONE}(x, y))$$

  Table 27 shows the truth table of the DC_ZERO operator.

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	1
1	1	0

Table 27. DC_ZERO operator truth table

- **NOT_ZERO_ONE**

  The NOT_ZERO_ONE operator takes two inputs, x and y, and returns false only if the values of x and y are false and true, respectively; otherwise, it returns true:

  $$\text{NOT\_ZERO\_ONE}(x, y) = 0$$

  It is equivalent to the negated form of the ZERO_ONE operator:

$$\text{NOT\_ZERO\_ONE}(x, y) \iff \neg(\text{ZERO\_ONE}(x, y))$$

Table 27 shows the truth table of the NOT_ZERO_ONE operator.

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	1
1	1	1

Table 28. NOT_ZERO_ONE operator truth table

- **ZERO_DC**

  The ZERO_DC operator takes two inputs, x and y, and returns true only if the value of the first input (i.e., x) is false, regardless of the value of the second input (i.e., y); otherwise, it returns false:

$$\text{ZERO\_DC}(x, y) = 1$$

  It is equivalent to the negated form of the ONE_DC operator:

$$\text{ZERO\_DC}(x, y) \iff \neg(\text{ONE\_DC}(x, y))$$

  Table 29 shows the truth table of the ZERO_DC operator.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	0
1	1	0

Table 29. ZERO_DC operator truth table

- **IMPLY**

  The IMPLY operator takes two inputs, x and y, and returns false only if the values of x and y are true and false, respectively; otherwise, it returns true:

$$x \implies y$$

  Table 30 shows the truth table of the IMPLY operator.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	0
1	1	1

Table 30. IMPLY operator truth table

- **NAND**

  The NAND operator takes two inputs, x and y, and returns false if the values of x and y are both true; otherwise, it returns true:

  $$x \overline{\wedge} y$$

  It is equivalent to the negated form of the AND operator:

  $$x \overline{\wedge} y \iff \neg(x \wedge y)$$

  Table 31 shows the truth table of the NAND operator.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table 31. NAND operator truth table

- **ONE** (binary)

  The binary ONE operator takes two inputs, x and y, and returns true, regardless of the values of both x and y:

  $$\text{ONE}_2(x, y) = 1$$

  The subscript "$_2$" is needed to distinguish between the binary ONE operator and the unary one.
  It is equivalent to the negated form of the binary ZERO operator:

  $$\text{ONE}_2(x, y) \iff \neg(\text{ZERO}_2(x, y))$$

  Table 32 shows the truth table of the binary ONE operator.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	1
1	1	1

Table 32. Binary ONE operator truth table

References

[1] Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides, Laura Pozzi, and Sherief Reda. *Approximate Logic Synthesis: A Survey*. 08 2020.

[2] Sherief Reda and Muhammad Shafique. *Approximate Circuits: Methodologies and CAD*. 2019.

[3] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. *Approximation-aware Rewriting of AIGs for Error Tolerant Applications*. 11 2016.

[4] Ashish Ranjan, Swagath Venkataramani, Shubham Jain, Younghoon Kim, Shankar Ganesh Ramasubramanian, Arnab Raha, Kaushik Roy, and Anand Raghunathan. *Automatic Synthesis Techniques for Approximate Circuits: Methodologies and CAD*. 01 2019.

[5] Simon Prince. *SAT Solvers I: Introduction and applications: https://www.borealisai.com/en/blog/tutorial-9-sat-solvers-i-introduction-and-applications/*.

[6] Linus Witschen, Tobias Wiersema, Matthias Artmann, Marco Platzner. *MUSCAT: MUS-based Circuit Approximation Technique*. 2022.

[7] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. *Approximation-aware Rewriting of AIGs for Error Tolerant Applications*. 2016.

[8] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. *BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization*. 05 2018.

[9] *Python website: https://www.python.org/*.

[10] University of California, Berkeley. *Berkeley Logic Interchange Format (BLIF)*. 07 1992.

[11] *Graphviz Documentation: https://graphviz.org/documentation/*.

[12] Jaroslav Bend and Ivana Cern. *MUST: Minimal Unsatisfiable Subsets Enumeration Tool*. 2020.

[13] *Yosys Documentation: https://yosyshq.net/yosys/documentation.html*.

[14] *Verilog Documentation: https://verilogguide.readthedocs.io/en/latest/*.

[15] *abc Documentation: http://people.eecs.berkeley.edu/~alanmi/abc/*.

[16] *Blif2GV Translator Repository Documentation: https://github.com/matteoalberici4/blif2gv-translator*.