Report on the Mixed Boolean-Algebraic Solver[1]

Edward Hirsch[2], Dmitry Itsykson[3], Arist Kojevnikov[2], Alexander Kulikov[2], Sergey Nikolenko[2]

basolver@logic.pdmi.ras.ru

Last changed: November 6, 2005

Abstract

We describe the basic notions and algorithm of the mixed boolean-algebraic solver being developed in the Laboratory of Mathematical Logic of St.Petersburg Department of Steklov Institute of Mathematics. The solver solves formulas of the propositional logic and checks boolean circuits for equivalence by translating them into systems of equalities and disjunctions of equalities and solving these systems by means of derivation rules together with traditional DPLL search.

The solver is implemented in `C++`. The implementation is flexible and allows to modify easily the rules employed by the proof system and even the nature of derivation objects.

[1]Supported by *Intel Corporation* via CRDF GAP project 1373(1).

[2]St.Petersburg Department of Steklov Institute of Mathematics, 27 Fontanka, 191023 St.Petersburg, Russia.

[3]Faculty of Mathematics and Mechanics, St.Petersburg State University, St.Petersburg, Russia.

Contents

Chapter 1

Introduction

The mixed boolean-algebraic solver solves the propositional satisfiability problem for formulas in conjunctive normal form (optionally, it also checks boolean circuits for equivalence and checks their properties described in a certain simple register-transfer level language). The original formula is transformed into a set of *objects*, which are either equalities or disjunctions of equalities. Then the solver executes a traditional DPLL [Davis and Putnam, 1960, Davis et al., 1962] search combined with a bunch of *rules* that generate new objects. The objects are held in *sets* that have indices that allow to find, add, and remove objects quickly. A *log* allows to undo changes before backtracking.

The main algorithm (Sect. 2.3) and the classification of derivation rules (Sect. 2.2) are implemented (and described) in a way that enables to use them with deduction and modification objects of any nature. However, the main idea of the mixed boolean-algebraic solver is the use of equalities. Accordingly, the specific types of objects we use are described in Sect. 3.1, and the specific rules are given in Sect. 3.3. A choice of the splitting heuristics is described in Sect. 3.4. The `C++` and command-line interfaces are described in Sect. 3.5. The numerical experimental data is given in Chap. 5.

The conclusion and advises for the usage or further development are given in Chap. 6.

The solver is implemented in the `C++` programming language, and the minimum amount of details concerning this implementation is given in the corresponding sections. The source code is quite flexible and allows

- implementation of solvers based on different mixed boolean-algebraic proof systems, in particular, solvers aimed at equivalence checking or verification of specific families of boolean circuits;
- implementation of solvers based on different (not mixed boolean-algebraic) objects;
- easy (even dynamic) replacement of data structures.

More implementation details are given in the source code and `doxygen` documentation.

Chapter 2

Abstract concept

2.1 Concept of the solver

The mixed boolean-algebraic solver (`basolver`) accepts either a formula in CNF, or a pair of boolean circuits, or a boolean circuit and a register-transfer level specification. It keeps a formula as a set of *deduction objects*; in the present solver these are *mixed boolean-algebraic clauses*, which are disjunctions of polynomial equalities in boolean variables. In addition to the set of deduction objects, we keep also a set of *modification objects* (a.k.a. *modifiers*). Most modifiers match one of the following types:

$$x = 0, \qquad \text{or} \qquad x = 1, \qquad \text{or} \qquad x = y, \qquad \text{or} \qquad x = 1 - y \ ,$$

where x, y are boolean variables; it is safe to forget about a modifier after it has been applied to every deduction object (until a satisfying assignment is to be generated). Another possible example of a modifier is an equality that determines the value of a variable as a function of other variables $x = f(y_1, \ldots, y_n)$, where x does not occur in deduction objects.

New objects are generated by means of derivation *rules* (described in Sect. 3.3). When a new object is derived, it is considered as a deduction object. We have a special rule that determines whether a clause (consisting of a single equality) deserves to become a modifier. All objects are stored in several sets in order to keep the invariant that each rule is applied to each group of objects at most once.

It is quite possible that the rules have already been applied in all possible ways, but we have not found neither a contradiction nor a satisfying assignment (the rules themselves do not form a complete set of deduction rules). In this situation we do a regular DPLL-like splitting; namely, we split the search process into two branches; in one of them we add a modifier $x = 0$ to the set of objects, and in the other branch we add $x = 1$ to the set of objects. Then we consequently process these branches, just like any other DPLL-type algorithm.

Similarly to all other DPLL-like solvers, we have a heuristic for choosing a literal (the specific one used in the current version is described in Sect. 3.4). We also have a heuristic for restricting the power of our rules in some nodes of the recursion tree (it is mentioned in Sect. 2.3, where the general abstract algorithm is described).

2.2 Rules classification

Rules are implemented as `C++` classes in the solver. The set of all rules except for the `TransferRule` is divided into two types: simplification and generation rules. Generation rules construct new objects from existing ones. Simplification rules are intended to replace existing deduction objects by new ones and to produce new modifiers. A simplification rule never increases the number of deduction objects, while a generation rule usually does. Due to this reason, simplification rules are called more often from the main algorithm. Both classes `SimplificationRule` and `GenerationRule` have a descendant whose name is obtained by adding a

prefix "Meta" to their name. The word "Meta" shows that a rule is in fact a list of several rules of one type. The application of this rule to a group of objects consists of consecutive applications of all its internal rules to this group. `TransferRule` is intended to divide newly produced objects into clauses and modifiers.

Below we describe the main interfaces of the rules used by `basolver`. By `DedObjSet*` and `ModObjSet*` we denote pointers to `DeductionObjectSet` and `ModificationObjectSet`, respectively. All rules are applied by means of their methods `operator()(...)`. Each such method (except for `TransferRule::operator()`) accepts a set of type `DedObjSet*` as a first parameter. It is the set for which all generated objects are added. Also, each rule is given a reference to the solver in the constructor. Some of the rules use this reference to check whether restrictions of their application are satisfied.

- `GenerationRule` is intended to generate new objects from existing ones.
  - `operator()(DedObjSet* returned_set, DedObjSet* new_set, DedObjSet* old_set)`

  Applies a rule to all possible groups of objects from `new_set` and `old_set` where at least one object is from `new_set`.

- `SimplificationRule` replaces its premises with new objects and/or generates new modifiers.
  - `operator()(DedObjSet* returned_set, DedObjSet* new_set, DedObjSet* old_set, bool new_vs_new)`
  - `operator()(DedObjSet* returned_set, ModObjSet* new_set, ModObjSet* old_set, bool new_vs_new)`

  These two operators apply a rule to all possible groups of objects from `new_set` and `old_set` where at least one object is from `new_set`; if the flag `new_vs_new` is set to False, then also at least one premise has to be from `old_set`.

  - `operator()(DedObjSet* returned_set, DedObjSet* ded_set)`

  This operator simplifies `ded_set` by means of itself; for example, the pure literal rule uses this interface to remove literals that do not occur negated in a set; since such rules use some global properties of variables which may not be extracted from any one set, in the algorithm we pass a list of several sets instead of one `ded_set` to this method.

- `TransferRule`
  - `operator()(DedObjSet* newset, DedObjSet* dset, ModObjSet* mset)`

  This operator is intended to move all objects from `newset` to either `dset` or `mset` according to the type of this object; thus, after the application of this method `newset` becomes empty.

2.3 Algorithm

During the algorithm run the following invariants are maintained:

1. No rule can be applied to a specific group of objects *more than once*.
2. If a rule has chances to be applied to a group of objects with a non-trivial result (when a new object is generated or an existing object is simplified), then it *has to be applied* to this group of objects or objects that resulted from their simplification (unless one of these objects disappears completely).
3. If the input formula is unsatisfiable, then at *every moment* of the algorithm execution the set of all deduction objects is unsatisfiable.

4. When a new object is generated, the algorithm checks whether this object is a modifier. If yes, it simplifies all existing objects by means of this modifier and puts it into the set `Modifiers`.

To satisfy the first two invariants at the preprocessing stage we apply the rules to the whole input sets, while at the main cycle stage a rule is applied to a set of objects only in case one of this objects was created at the previous iteration of the cycle.

Algorithm:

Input: Set F of deduction objects.
Output: Satisfying assignment for F, if any; `False`, otherwise.

Method.

1. **Initializing steps:**

   `Clauses` $= \emptyset$.

   `Modifiers` $= \emptyset$.

   `Clauses = NewObjects = NewClauses` $= \emptyset$.

   `RecentClauses = clauses of` F.

   if `RecentClauses` contains modifiers, apply them and put them into `Modifiers`.

2. **Main Cycle:**

   (a) do

      i. do

         A. `MetaSimplificationRule(NewObjects, Clauses` $\cup$ `NewClauses` $\cup$ `RecentClauses);`

         B. `MetaSimplificationRule(NewObjects, RecentClauses, NewClauses, false);`

         C. `MetaSimplificationRule(NewObjects, RecentClauses, Clauses, true);`

         D. move all objects from `RecentClauses` to `NewClauses`;

         E. move all objects from `NewObjects` to `RecentClauses`;

      while `RecentClauses` $\neq \emptyset$.

      ii. if the current depth of the search tree is equal to $1, 2, 4, \ldots, 2^k, \ldots$

         A. `MetaGenerationRule(NewObjects, NewClauses, Clauses);`

         B. move all objects from `NewClauses` to `Clauses`;

         C. move all objects from `NewObjects` to `RecentClauses`;

   while `RecentClauses` $\neq \emptyset$.

   (b) if `Clauses` is empty, return satisfying assignment

   (c) Select a variable x for splitting according to some heuristic. Split the current formula on this variable (in one branch we simplify all objects by an equality $x = 0$ and put this equality to the set `Modifiers`, in the other we do the same with $x = 1$) and recursively call for the Main Cycle on both constructed formulas. If at least one of the recursive calls returns a satisfying assignment, update this assignment (by processing modifiers from `Modifiers`) and return it; otherwise return `False`.

Chapter 3

Specific solver

3.1 Objects and notation

The `basolver` keeps its objects as sets of deduction objects (of type `DeductionObject`) and modification objects (of type `ModificationObject`). A deduction object is a disjunction (clause) of several equalities consisting of boolean variables (an element of such a disjunction is called `SALiteral` in the implementation). A clause becomes a modifier in the following two cases: either it consists of a single equality of type $x = \{0, 1\}$ or $x = \{y, 1 - y\}$, where x, y are boolean variables (this condition is checked by a special rule called `TransferRule`), or it is generated by either `SingleOccurrenceRule` or `PureLiteralRule`. Each time a new modifier is generated, the solver simplifies all current objects by this modifier and moves it to the set `Modifiers`.

Equalities (type `Equality`) have the form LHS=RHS, where LHS and RHS are polynomials (of type `Polynomial`), that is, sums of monomials. Below we describe the invariants of these structures that need to be satisfied for the program to work correctly. For example, a natural invariant for a class representing polynomial is that is does not contain equal monomials.

A monomial (type `Monomial`) is a product of an integer coefficient and several boolean variables. Since we have $x^2 = 1$ for any boolean variable x, the set of variables of a monomial does not contain equal variables (that is, it is truly a set, not a multiset). We use the following monomial ordering:

- two monomials are equal if they have equal sets of variables (but their coefficients may be different);
- if two monomials have different degrees, the one with smaller degree is smaller;
- if two monomials have equal degrees but different sets of variables we use the lexicographical order; e.g., $p_1p_2p_6$ is smaller than $p_1p_3p_4$.

Equality invariants are given below. We say that an equality is normalized, if it satisfies all these invariants.

- LHS contains only one monomial, the coefficient of this monomial is positive, and the monomial is not a constant.
- LHS and RHS do not share equal monomials.
- The most common divisor of all monomial coefficients is 1.
- An equality cannot have the form $a = -b + c + 2ab$, where a, b, c are literals corresponding to different variables (if it has such a form before normalization, it is transformed to the equivalent form $x = y + z - 2yz$, where the sets $\{a, b, c\}$ and $\{x, y, z\}$ are equal). We call this operation *XOR-rewriting*.
- LHS contains the minimal monomial among monomials with the minimal absolute value of coefficient.

Each time a new object is generated, the solver does the following:

1. Brings a new object to a canonic form by means of `StrongNormalizeRule`:

$$\frac{x = 2P}{x = 0 \qquad 0 = P}, \qquad \frac{x = l + 2P}{x = l \qquad 0 = P}, \qquad \frac{x = 1 + 2P}{x = 1 \qquad 0 = P}, \qquad \frac{x = -y + 2S}{x = y \qquad 0 = -y + S},$$

$$\frac{0 = n - \sum_{i=1}^{i<n} m_i + \sum_j m_j}{0 = 1}, \qquad \frac{0 = n - \sum_{i=1}^{n} m_i + \sum_j m_j}{m_i = 1 \qquad m_j = 0},$$

   where P is a polynomial, x, y are variables, n is a positive integer, m_i, m_j are non-constant monomials with coefficient 1.

   Also, if a new object contains less than four variables, this rule finds all its satisfying assignments and creates an equality $l = 1$ or $l_1 = l_2$, where l, l_1, and l_2 are literals, if this equality holds for all these assignments. For example, given an equality $2x = 2 - 2y + xy$ it produces an equality $x = 1 - y$.

2. Checks whether a (simplified) new object is a tautology or a contradiction.

3. If a (simplified) new object is a modifier, the solver simplifies all objects by means of this modifier. (this is implemented in the class `LiteralSubstitutionRule` that is designed to simplify objects by modifiers) and puts it into the set `Modifiers`.

Throughout the whole project we use `P` in front of the name of a class to denote a *smart pointer* type for objects of that class. The smart pointers come from the *Boost* package, which we have used in the project. Further information concerning the Boost classes and methods may be found at `http://www.boost.org`.

3.2 Basic C++ classes

Below we describe the main classes of `basolver`. Note that most classes have several invariants that need to be satisfied for the `basolver` to work correctly. So, whenever you are modifying a method of a class do not forget to check that your modification preserves all invariants of this class. See `doxygen` documentation for invariants description. (Actually the `doxygen` documentation is more convenient to read as it contains links to the detailed description of each class mentioned in the text).

For most classes we have abstract interfaces. This is done for easy substitution of implementations of an interface. Thus, instantiations are made inside generators of `basolver` objects of specific implementations. Usually interfaces are implemented in a class whose name is obtained by adding prefix "Simple" to the name of an interface (notable exceptions are `TypedEquality` and `BooleanEquality` that are currently the only implementations of Equality).

3.2.1 Algebraic classes

Algebraic classes are used to construct mixed boolean-algebraic extensions of logical classes.

- `Coefficient` is just an integer coefficient.
- `Variable` is what a typical boolean variable is.
- `Monomial` is a product of `Coefficient` and several variables (note that since we work with boolean variables this product contains each variable at most once).
- `Polynomial` is a sum of monomials.
- `Equality` is an entity of the form $poly_1 = poly_2$, where $poly_1$ and $poly_2$ are polynomials (see also Sect. 3.1).
- `AlgebraicGenerator` is a generator of algebraic objects of specific implementations.

3.2.2 Logical classes

Logical classes represent mixed boolean-algebraic extensions of typical logical objects, such as a boolean literal or a boolean clause.

- `SALiteral` is a mixed boolean-algebraic extension of a boolean literal. Currently, only an equality (of type `Equality`) may be used as a `SALiteral`.
- `SAClause` is a mixed boolean-algebraic extension of a boolean clause. Thus, `SAClause` is a disjunction of several `SALiterals`. For example, a boolean clause $(x_1 \vee x_2 \vee \overline{x_3})$ has the following representation in the `basolver`: $(x_1 = 1 \vee x_2 = 1 \vee x_3 = 0)$.
- `LogicalGenerator` is a generator of logical objects of specific implementations.

Note that we do not have a class corresponding to the notion of a CNF formula. We use sets of objects instead (of template type `ObjectSet`, instantiating it to contain either `DeductionObject` or `ModificationObject`).

3.2.3 Sets

Any object of the search space is considered either as a deduction object (of type `DeductionObject`) or a modification object (of type `ModificationObject`). Modification objects are those by means of which we simplify other objects. As modification objects we consider only equalities $l = 0$ and $l_1 = l_2$, where l, l_1, and l_2 are literals and equalities generated by `PureLiteralRule` and `SingleOccurrenceRule`. These objects are stored in sets which are respective localizations of the `ObjectSet` template. The standard implementation we use is `SimpleObjectSet` which has several internal indices (such as, for example, adjacency lists). These indices are kept in a special structure (`SimpleObjectIndex`). However, for certain sets which do not require indices at all we use `TrivialObjectSet` — the simplest set possible with cheap basic operations. Both sets allow for all typical set operations (such as `getSize()`, `add()`, `clear()`) and also operations on iterators with sets (`getBeginIteratorOnWholeSet()`, `getEndIteratorOnWholeSet()`, `remove(PSimpleObjectIterator)`). Note that all operations with iterators are in fact implemented in indexing classes (`SimpleObjectIndex`).

The `SimpleObjectSet` on `DeductionObject` is called `AdjDedObjSet`. The `SimpleObjectSet` on `ModificationObject` is called `AdjModObjSet`.

3.2.4 Indexing classes

These classes represent indices and iterators of `basolver`. Index is a structure for storing objects of a set intended to quicker perform the operation of extracting objects that satisfy some predefined condition. For example, when we want to assign the value 1 to a variable x in a formula, we only have to change objects that contain x. The corresponding index (adjacency list) returns the necessary set of clauses, and we do not need to traverse all clauses in the set. An iterator is an entity allowing to iterate conveniently on all selected objects.

We have indices and iterators for several objects of the program (we use our own indices and iterators for easier substitution of implementations; however, our structures are often just wrappers for standard `C++` libraries indices and iterators).

The `SimpleObjectSet` class allows to take the following iterators:

- `SimpleObjectSet::getBeginIteratorOnWholeSet()` returns an iterator on the whole set;
- `SimpleObjectSet::begin(Variable)` returns an iterator on all objects of the set containing the variable given as parameter;
- `SimpleObjectSet::getLHSbegin(Variable)` returns the begin iterator on all equalities of the set whose left-hand side contains the given variable.

- `SimpleObjectSet::getBegin(TIndexType)` returns the begin iterator on all equalities of the set that belong to the index given as parameter. The enumerative type `TIndexType` corresponds mostly to equality types (enumerative type `TEqualityType`) and is declared in `src/basicobject/object.h`.

See object types description for further information on the objects this index contains.

To create an iterator for the end of the corresponding list, use `...End...` instead of `...Begin...` in each of the cases described above.

- `SimpleObjectSet::getFirstObjectInVars()`. This index is supported somewhat differently; its current iterator is kept inside the indexing system. Use this method for getting the first object whose variables are contained in the given three. Use `SimpleObjectSet::nextObjectInVars()` for traversing the set of such objects, `SimpleObjectSet::currentObjectInVars()` for peeking at the current object, and `SimpleObjectSet::isLastObjectInVars()` for testing whether you are at the end of the list.

All these indices are supported in `SimpleObjectSet` and are not supported in `TrivialObjectSet` (`ObjectSet` has an assertion against their use in the trivial implementation).

3.2.5 Backtracking classes

These classes are intended to undo modifications made during the proof search process. This is also known as backtracking and used in all DPLL-like algorithms. The program has the class `Log` and `BooleanAlgebraicSolver` has a member `Log* myLog`. This member contains all necessary information needed to undo modifications.

To undo all modifications made after some moment of time one has to remember this moment of time (this is implemented in the class `Bookmark`) and then (after applying modifications) take this moment to the solver's `Log`. However there are sets that do not require to be returned to the state of this moment (for example, temporary sets). To indicate whether or not all modifications of a set have to be remembered in a `Log` use the flag `myWritesLog` (access through `getWriteLog()` and `setWriteLog(bool)`).

3.2.6 Rules classes

These classes implement simplification and generation rules of the `basolver`. A simplification rule either replaces deduction objects with new ones or generates new modifier. A generation rule produces new objects from existing objects. Thus, a simplification rule never increases the number of deduction objects, while a generation rule usually does. We also have a specific rule (`TransferRule`) which is intended to distinguish deduction and modification objects. See Sect. 3.3 for specific rules description.

3.2.7 Main algorithm

The main algorithm is implemented as a `BooleanAlgebraicSolver::solve()` method (in file `src/general/mainalgorithm.cc`). Rules can be added to meta-rules of the algorithm in the constructor `BooleanAlgebraicSolver::BooleanAlgebraicSolver()`. See Sect. 2.3 for the main algorithm description.

3.3 Specific rules used by the solver

3.3.1 Notation

We use the following notation in the premises and conclusions of the rules described below:

- Zero in the left hand side of an equality means that it is *not* given in the normal form, while in any other case it is supposed that the equality is normalized.
- Different lower-case letters mean literals of different variables if not explicitly stated otherwise.

- We assume that all equalities that have only lower-case letters in their descriptions are weakly normalized in the following sense: all equality invariants (described in Sect. 3.1) are true except XOR-rewriting.

We also freely use the terms *odd part* (resp. *even part*) of a polynomial. They mean the part of this polynomial that contains its monomials with odd (resp. even) coefficients. When we want to indicate a certain rule, we write the name of its class rather than the name of the rule itself; this makes explanations more clear.

3.3.2 Equality types and rule applications

The implementation uses so-called *equality types* that help to save time on recognizing whether a given equality has certain form (e.g. $x = ab + ac - abc$, where x, a, b, c are literals). Rules use equality types to determine whether the given equality may be a premise. However, only basic types (namely $x = ab$, $x = a + b - 2ab$, and $ab = 0$, where x, a, b are literals) are guaranteed to be recognized, that is, if an equality has this form, it has the corresponding type. Other types are given only to equalities that are produced by the corresponding rule applications, and if an equality assumes a certain form after some other process (like substituting a monomial with a polynomial or summing up two equalities), it may not be recognized as having this form and, consequently, will not be subject to rules that accept these equalities as premises.

Not every rule requires typed equalities as input or produces typed equalities as output. In what follows we mark premises that should be typed and results that are created as typed equalities with $^{\texttt{typed}}$. If an equality is not marked with $^{\texttt{typed}}$, it has type `Special`.

Please refer to Sect. 3.6 for the tables that explain how different types may transfer into each other and what symmetries should be taken into account while processing a typed equality.

3.3.3 Generation rules

A generation rule generates new objects from given object(s) or set(s). Usually, a generation rule does not remove any of its premises. Almost all generation rules of `basolver` are particular cases of the following **General Substitution Rule**:

$$\frac{0 = S \qquad 0 = -x + P}{0 = S|_{x=P}}$$

This rule just substitutes a variable x by a polynomial P in an equality $0 = S$ (it is supposed that x does not occur in P). However, we cannot use this rule without restrictions as it produces too many objects. Therefore we have several particular cases of this rule, which differ only by conditions that need to be satisfied by the equalities $0 = S$ and $0 = -x + P$. Below we list these "subrules" of the General Substitution Rule (the names of the rules sometimes contain something like "XOR3"; this means "the XOR function of 3 literals"). By the $^{\texttt{typed}}$superscript we indicate that this equality has a certain type; if the type is not obvious, we mention it in the description.

1. **AND Substitution**

   Class: `BackLinearization2Rule`

   Description:

   $$\frac{0 = -x + yz^{\texttt{typed}} \qquad 0 = -y + S}{0 = -x + Sz}.$$

   The rule is applied only if the sets of the solver do not contain equalities like $0 = z - ll'$ and $0 = -z + x + y - 2xy$, and the second premise is linear, has an even coefficient, contains not more than 5 variables, and S does not contain y.

2. **Summation — second part**

   Class: `Summation2Rule`

Description:

$$\frac{0 = -x + a + bc - ac^{\texttt{typed}} \qquad 0 = a - b + S}{0 = -x + a + cS}$$

where x, a, b, c are literals. The rule is applied only in case the second premise is linear, has an even coefficient, contains not more than 5 variables, and S does not contain b. First premise has the type `eqtYeqACpnAB`.

3. **OR3 generation**

   Class: `OR3GenerationRule`

   Description:

$$\frac{0 = -d + ab^{\texttt{typed}} \qquad 0 = -y + c + d - cd^{\texttt{typed}}}{0 = -y + ab + c - abc^{\texttt{typed}}}, \qquad \frac{0 = -d + ab^{\texttt{typed}} \qquad 0 = -y + ac + d - acd^{\texttt{typed}}}{0 = -y + ac + ab - abc^{\texttt{typed}}},$$

$$\frac{0 = -d + ab + ac - abc^{\texttt{typed}} \qquad 0 = -y + bc + d - bcd^{\texttt{typed}}}{0 = -y + ab + ac + bc - 2abc^{\texttt{typed}}},$$

   where some of the literals could be equal to each other (however, $a \neq \{b, \neg b\}$, $c \neq \{d, \neg d\}$ in the first part, $b \neq \{c, \neg c\}$ in the third part).

4. **OR3 generation — second part**

   Class: `OR3GenerationRule2`

   Description:

$$\frac{0 = -d + (1-a)b^{\texttt{typed}} \qquad 0 = -y + ac + d - acd^{\texttt{typed}}}{0 = -y + ac + (1-a)b^{\texttt{typed}}},$$

   where some of the literals could be equal to each other (however, $a \neq \{b, \neg b\}$).

5. **Substitutions of 5 Variables to XOR — first part**

   Class: `XOROR2Part1Rule`

   Description:

$$\frac{0 = -g + y - yw^{\texttt{typed}} \qquad 0 = -c + yg + zg - 2yzg^{\texttt{typed}}}{0 = -c + y - wy - yz + yzw^{\texttt{typed}}},$$

$$\frac{0 = -g + yw^{\texttt{typed}} \qquad 0 = -c + yg + zg - 2yzg^{\texttt{typed}}}{0 = -c + wy - yzw^{\texttt{typed}}}.$$

   Note that the types of the result of the two descriptions of this rule are identical (`eqtZeqWXpVXm2WVX`): the visible difference stems from the different literal signs the two equalities employ.

   The rule is restricted in the following way: if the `--booth` option is not given (proofs for the Booth multipliers do not have this property), the rule may be applied only if at least one of the variables of the second premise (the XOR-encoding equality) also occurs in an equality that has already been deleted by `SingleOccurrenceRule`.

   Remark 3.3.1. *This rule has the following usage. During the process of proving equivalences between two multipliers we occasionally run into the situation when two outputs of two full adders have been proven equal, and their inputs as well. In this case,* `basolver` *rules would make the two equalities describing the last XOR functions of these full adders identical, and they would be subsumed against each other. This interferes with the normal inference. This rule provides an "intermediate" representation for this XOR combined with the OR it is connected with, and thus allows to continue inference in the correct way.*

In addition, we have a few rules that do not match the form of **General Substitution Rule**. They either somehow change the first premise before substituting a variable in it or somehow simplify the result.

1. **Partial XOR3 Linearization**

   Class: `LinXorRule`

   Description:

$$\frac{v = xy + xz + yz - 2xyz^{\texttt{typed}} \qquad 0 = -l_w + l_x l_y + l_x l_z - 2 l_x l_y l_z^{\texttt{typed}}}{0 = -v + xy + xz + yz - 2xyz + \alpha(-l_w + l_x l_y + l_x l_z - 2 l_x l_y l_z)},$$

   l_t denotes a literal of a variable $t \in \{v, w, y, z, x\}$, the coefficient $\alpha \in \{1, -1\}$ is selected so that the conclusion has degree 2.

2. **Linearization for Functions of 2 Variables**

   Class: `Linearization2Rule`

   Description:

$$\frac{0 = -x + ab^{\texttt{typed}} \qquad y = a + b - 2ab^{\texttt{typed}}}{0 = -y + a + b - 2x}, \qquad \frac{0 = -x + ab^{\texttt{typed}} \qquad a = y + b - 2yb^{\texttt{typed}}}{0 = -y + a + b - 2x}.$$

   The rule does not apply if the variable corresponding to y occurs in exactly one equality (except for the premise), and this equality encodes the representation $y \oplus z \oplus t = \texttt{const}$, where variables corresponding to y, z, and t do not occur in boolean clauses.

   The resulting equality has type `Special`.

3. **Summation**

   Class: `SummationRule`

   Description:

$$\frac{y = S \qquad y = T}{0 = S - T}, \qquad \frac{y = S \qquad u = y + T}{0 = -u + S + T}.$$

   The rule describes adding two equalities with canceling out a linear monomial. The following restrictions on the premises and the result currently apply:

   (a) both premises consist of monomials of degree ≤ 2;

   (b) coefficients of degree 2 monomials in both premises are even;

   (c) the equality $y = S$ is linear;

   (d) the equalities $y = T$ and $u = y + T$ are either both linear or have degree 2 and contain a monomial occurring in clauses of size ≥ 2 (that is, occurring in a boolean clause that has several literals); this restriction is lifted if the result of the possible rule application is contradictory;

   (e) in the equality $u = y + T$ the monomial y is minimal among all monomials that do not occur in clauses of size ≥ 2, while the monomial u occurs in a clause of size ≥ 2;

   (f) all coefficients in the conclusion are not bigger then 2 in absolute value;

   (g) if the conclusion is linear and does not contain even monomials then the number of variables in it should be equal to the number of monomials in even parts of T and S, both even parts of T and S are not empty;

   (h) at least one of the premises must contain at least one monomial with an even coefficient;

   (i) if the conclusion is linear and contains even monomials then monomials in the odd part of the conclusion should not contain variables from even parts of T and S;

   (j) if the conclusion is linear and contains even monomials then the size of even part should be equal to sum of even part sizes of T and S;

(k) if the right-hand side of the conclusion is linear and contains even monomials then the variable in the left-hand side of the conclusion should not be from even parts of T and S;

(l) if the conclusion is linear then the sets of variables occurring in the even parts of T and S are disjoint.

4. **XOR Summation**

   Class: `XorSummationRule`

   Description:

$$\frac{y = S^{\texttt{typed}} \qquad y = T^{\texttt{typed}}}{0 = S - T^{\texttt{typed}}},$$

   We currently have the following restrictions on the sums of monomials of S, T, and the result $S - T$:

   (a) the result has degree 2;

   (b) either both sums have degree 2, or one of the equalities is linear and the other has type `eqt11m2`;

   (c) coefficients of the degree 2 monomials are even;

   (d) each premise has type `eqt11m2`, `eqtXORSum`, `eqtXORSumWithLin`, or is linear and has type `Special`.

   The first premise with type `eqtXORSum` or `eqtXORSumWithLin` and the premise with the greatest ID (that is, the premise generated last) is removed. The result has the type `eqtXORSum` unless one of the premises is linear or is already of type `eqtXORSumWithLin`; in the latter case the result also obtains type `eqtXORSumWithLin`.

3.3.4 Simplification rules

A simplification rule either replaces input objects by new ones or generates new modifiers, thus, a simplification rule never increases the number of deduction objects. Simplification rules achieve that by deleting some of their premises. We further denote by (kept) the premise that is not deleted after a simplification rule has been applied.

1. **Back And Substitution**

   Class: `BackT2SubstitutionRule`

   Description:

$$\frac{a = xy + S \qquad c = lg^{\texttt{typed}}\ \text{(kept)}}{0 = -a + S'},$$

   where l and g are literals of variables x and y, S and S' are sums of monomials. The conclusion of the rule does not contain the monomial xy with any coefficient. The first equality has degree 2 and type `Special`. Otherwise, the rule does not apply.

2. **Binary Clauses Processing**

   Class: `BinaryClauseProcessingRule`

   Description: Processes the 2-CNF part of a formula by the following algorithm: if setting $x = 1$ makes the 2-CNF part unsatisfiable, we can assign $x = 0$ (see [del Val, 2000] for more details). Also, replaces two clauses — $(x \vee y)$ and $(\bar{x} \vee \bar{y})$ — by an equality $x = 1 - y$. (Note that a 2-clause $(x \vee y)$ may be represented as $x + y - xy = 1$.)

3. **Gates Equivalence**

   Class: implemented as a part of the `GatesEquivalence2Rule` class

Description:

$$\frac{0 = S \qquad 0 = T}{x = l \qquad 0 = S|_{x=l} \qquad 0 = T|_{x=l}},$$

given that $S + T$ may be rewritten as $x + l - 2P$, where S, T, and P are polynomials, x is a variable, l is a literal or a constant 0 or 1. The rule is applied only in the case when either both input equalities have the same type and this type is `Special`, `eqtXeqAB`, `eqt11m2`, `eqtXeqABpACpBCm2ABC` or `eqtBoothSubResult` or one of them is of type `Special` and the other is of type `eqtBoothSumResult`. If they have type `Special`, the rule is applied only if the sum of their degrees is equal to 3.

4. **Equivalence of Even Parts**

   Class: implemented as a part of the `GatesEquivalence2` class

   Description:

$$\frac{0 = L + 2S \text{ (kept)} \qquad 0 = L + 2T}{0 = S - T}.$$

   If at least one of the premises is linear, it should be kept. The polynomials S and T have degree ≤ 2. The rule does not apply if S and T are both literals (otherwise with some renaming of variables of the input formula we may cancel out the carry bit sums of two circuits on the same level). The rule does not apply if the odd part of $S - T$ contains a monomial of degree 2 (otherwise the result of `LinearizationRule2` would remove its own premise). However, it should be applied in the latter case, if the resulting equality looks like $x = ab$ (has type `eqtXeqAB`).

5. **Linearization for Functions of 3 Variables**

   Class: `Linearization3Rule`

   Description:

$$\frac{y = a + b + c - 2ab - 2ac - 2bc + 4abc^{\texttt{typed}} \qquad 0 = -x + ab + ac + bc - 2abc^{\texttt{typed}} \text{ (kept)}}{0 = -y + a + b + c - 2x},$$

$$\frac{a = y + b + c - 2yb - 2yc - 2bc + 4ybc^{\texttt{typed}} \qquad 0 = -x + ab + ac + bc - 2abc^{\texttt{typed}} \text{ (kept)}}{0 = -y + a + b + c - 2x}.$$

6. **Monomial Substitution:**

   Class: MonomialSubstitution

   Description:

$$\frac{0 = r + p \cdot xy \qquad 0 = xy + \alpha x + \beta y + \gamma^{\texttt{typed}} \text{ (kept)}}{0 = r + p(-\alpha x - \beta y - \gamma)}, \tag{3.3.1}$$

   where r and p are polynomials, x, y are variables, r does not contain xy, α, β, γ are integers and the second premise is of type `eqtABeq0`.

7. **Pure Literal**

   Class: `PureLiteralRule`

   Description:

   This rule acts as a classical pure literal rule: it assigns literals that occur only positively or only negatively. Since it is unclear how to distinguish positive and negative occurrences in non-boolean equalities, `basolver` applies this rule only if the variable in question does not occur in non-boolean clauses.

8. **Single Occurrence**

   Class: `SingleOccurrenceRule`

   Description:

   Suppose that a variable occurs in only one clause in the whole formula.

   This rule is aimed at removing such occurrences. If it encounters a variable that occurs in only one clause, and this clause is a Boolean clause or an equality of type `eqtXeqAB` or `eqt11m2`, it moves that clause to `Modifiers`. Basically, this is equivalent to removing the clause. The only difference is that when `basolver` calculates a satisfying assignment, it uses modifiers from the `Modifiers` set to calculate missing variables. Otherwise the resulting assignment would be incorrect or incomplete.

9. **Functions Extraction**

   Class: `CircuitFormTranslationRule`

   Description:

   Input: Set of clauses.

   (a) We assume that Subsumption, Binary Clauses Processing, have been already applied. For each variable a in the clause list we find a clause (l_a, l_b, l_c) containing it.

   (b) If list of clauses contains clauses $(\neg l_a, \neg l_b)$ and $(\neg l_a, \neg l_c)$ add to the set the normal form of the following functional representation:

   $$l_a = (\neg l_b) \cdot (\neg l_c).$$

   If x is a variable the functional representation of literal x is x, $\neg x$ is $1 - x$. Delete clauses (l_a, l_b, l_c), $(\neg l_a, \neg l_b)$ and $(\neg l_a, \neg l_c)$.

   (c) If list of clauses contains clauses (l_a, l_b, l_c), $(l_a, \neg l_b, \neg l_c)$, $(\neg l_a, l_b, \neg l_c)$ and $(\neg l_a, \neg l_b, l_c)$, add to the set the normal form of the following representation:

   $$\neg l_a = \neg l_b + \neg l_c - 2\neg l_b \neg l_c.$$

   Delete clauses (l_a, l_b, l_c), $(l_a, \neg l_b, \neg l_c)$, $(\neg l_a, l_b, \neg l_c)$ and $(\neg l_a, \neg l_b, l_c)$.

   Output:

   (a) boolean circuit representation in form of equalities $o = f(i_1, \ldots, i_m)$. Here o is the sub-circuit output, i_j denote inputs, f is a polynomial of degree ≤ 2.

   (b) unconverted clauses.

   Remark 3.3.2. *This is a primary rule for generating non-linear (namely, quadratic) equalities. It extracts unary and binary functions from boolean* 2- *and* 3*-clauses.*

10. **Simple Subsumption**

    Class: `SimpleSubsumptionRule`

    Description:

    This rule performs the usual subsumption: delete from a set every SAClause that is a subset of another SAClause.

3.3.5 Rules specific for Booth multipliers

Not all rules are used in all verification proofs. The following five rules are used only in proving correctness of Booth multipliers (in our case, only simplified Booth so far). If used in every proof search, they would clutter the solver with redundant equalities (with high degree and a lot of monomials in them), so these rules are turned off by default. To turn these rules on, use the `--booth` command-line option. All these rules have type `GenerationRule` except for the `XOROR2Part3` rule.

1. **Back T2 Substitution**

   Class: `BackT2SubstitutionRule`

   Description:

   This rule is described in Subsect. 3.3.4. With `--booth` option it works differently: it is applied if the first premise has type `eqtBoothSubResult` and in this case the first premise is not removed.

2. **Substitution of Booth Numbers to Linear Sum**

   Class: `BoothSubRule`

   Description:

$$\frac{0 = c - S^{\texttt{typed}} \qquad 0 = T + \alpha c}{0 = T + \alpha S^{\texttt{typed}}}, \qquad \frac{0 = c - S^{\texttt{typed}} \qquad 0 = T + \alpha c^{\texttt{typed}}}{0 = T + \alpha S^{\texttt{typed}}},$$

   where equality $0 = c - S$ is one of the following:

$$0 = -c + xz - xy + y - wy + 2xwy - yz + yzw - 2yzxw,$$

$$0 = -c - xy + y - wy + 2xwy, \quad 0 = -c + y - wy - yz + yzw,$$

   $0 = T + \alpha c$ is either a linear equality generated from Full Adder or an equality generated by this rule, and v, w, y, z, x, are different literals.

   To ensure that the rule does not generate excessive amounts of unnecessary equalities, the conclusion of this rule contains a list of literals that should be deleted from this conclusion by the following applications of this rule. This list is initialized with variables occurring in the linear representation of a Full Adder, except for the deleted literal. As a result, the representations of negative numbers are step by step substituted into the linear equalities that represent a Full Adder on the current level, and then are summed up together.

   We impose the following restrictions on this rule:

   (a) the rule applies only if both objects have type `eqtBoothSubResult` or `Special`, or if the second object is not of the type `eqt11m2`;

   (b) if the first variables of the literals lists of the two premises differ, the second variables are 0, or the second object has type `eqtXeqAB`, then the rule applies only if the literals lists of the odd parts of both equalities have more than two variables in common;

   (c) if the second object has type `eqt11m2`, and its degree 2 monomial occurs in the even part of the first object, the rule does not apply;

   (d) if the second object has type `eqt11m2`, it should be matched up only with such objects of type `eqtBoothSubResult2` or `eqtBoothSubResult` that have 0 as the second literal in their literal lists;

   (e) do not apply the rule if one of the premises is of type `eqtBoothSubResult` with an empty literal list;

   (f) if both objects have type `eqtBoothSubResult`, and the second variables in their literal lists are 0, do not apply the rule;

   (g) if one of the premises has type `eqtBoothSubResult` and the second premise has type `Special` or `eqtBoothSubResult`, and the third literals in their literal lists are not 0, do not apply the rule;

   (h) if S contains odd monomials, then the monomials in even part of T should not occur at the same time in odd parts of S.

3. **OR3 substitution to OR**

   Class: `ORORRule`

Description:

$$\frac{0 = -v + xyz^{\texttt{typed}} \qquad 0 = -x + ls^{\texttt{typed}}}{0 = -v + yzls^{\texttt{typed}}}$$

$$\frac{0 = -v + xyz^{\texttt{typed}} \qquad 0 = -1 + x + ls^{\texttt{typed}}}{0 = -v + yz - yzls^{\texttt{typed}}}$$

where v, x, y, z, l, s are different literals.

4. **OR2 substitution to XOR3**

   Class: `ORXOR3Rule`

   Description:

$$\frac{0 = -y + wv^{\texttt{typed}} \qquad s = x + y + w - 2xy - 2xw - 2yw + 4xyw^{\texttt{typed}}}{0 = -s + x + w - wv + 2xwv - 2xw^{\texttt{typed}}},$$

$$\frac{0 = -y + wv^{\texttt{typed}} \qquad y = x + s + w - 2xs - 2xw - 2sw + 4xsw^{\texttt{typed}}}{0 = -s + x + w - wv + 2xwv - 2xw^{\texttt{typed}}}.$$

5. **Substitution of 5 Variables to XOR — second part**

   Class: `XOROR2Part2Rule`

   Description:

$$\frac{0 = -z + xy^{\texttt{typed}} \qquad y = w + v - 2wv^{\texttt{typed}}}{0 = -z + wx + vx - 2wvx^{\texttt{typed}}}, \qquad \frac{0 = -z + xy^{\texttt{typed}} \qquad v = w + y - 2wy^{\texttt{typed}}}{0 = -z + wx + vx - 2wvx^{\texttt{typed}}},$$

$$\frac{0 = -g + x + y - yw + 2xwy - 2yx^{\texttt{typed}} \qquad 0 = -c + yg + zg - 2yzg^{\texttt{typed}}}{0 = -c + xz - xy + y - wy + 2xwy - yz + yzw - 2yzxw^{\texttt{typed}}},$$

$$\frac{0 = -x + g + y - yw + 2gwy - 2yg^{\texttt{typed}} \qquad 0 = -c + yg + zg - 2yzg^{\texttt{typed}}}{0 = -c + xz - xy + y - wy + 2xwy - yz + yzw - 2yzxw^{\texttt{typed}}}.$$

6. **Substitution of 5 Variables to XOR — third part**

   Class: `XOROR2Part3Rule`

   Description:

$$\frac{0 = -g + x + y - yw + 2xwy - 2yx^{\texttt{typed}} \qquad 0 = -c + yg^{\texttt{typed}} \text{ (kept)}}{0 = -c - xy + y - wy + 2xwy^{\texttt{typed}}},$$

$$\frac{0 = -g + x + y - yw + 2xwy - 2yx^{\texttt{typed}} \qquad 0 = -c + yx^{\texttt{typed}} \text{ (kept)}}{0 = -c - gy + y - wy + 2gwy^{\texttt{typed}}}.$$

   Note that this is the only Booth-related rule that deletes its premise.

3.4 Splitting heuristic

In this section we deal with a basic question for all DPLL-like solvers: how to choose the next variable to split on. Various heuristics have been suggested and tested during our project. We present here the final version, which we hardly can back up theoretically, but rather have selected on the experimental basis.

In the present version of the solver we use the following heuristics for choosing a variable for splitting (a smaller number in the list means a higher priority):

1. Never split by a variable that does not occur at all.

2. Some derivation rules set priorities for the variables that occur in the conclusion and have not yet been assigned with a priority. These priorities are further used by the following splitting heuristics.

   Choose a variable with higher priority that is set by derivation rules. At the moment, the rules set:

   - `BackT2SubstitutionRule`: -10 (the highest),
   - `Linearization3Rule`: -1,
   - `GatesEquivalence2Rule` (Equivalence of even parts): 1,
   - `GatesEquivalence2Rule` (Gates equivalence): 10 (the lowest);

   the default priority is 0, and, once set, the priority is not changed until a substitution of a modifier $x = y$ or $x = 1 - y$ occurs, in which case the higher priority of x and y is chosen for x.

Let c_k be the number of occurrences in boolean k-clauses, and c_* be the number of occurrences in boolean clauses. Obviously, the more often a variable occurs, the more desirable it is in general for splitting. We have made experiments with different weights and came up with the following heuristics.

3. Choose a variable with the highest $4c_2 + 2c_3 + c_*$.
4. Choose a variable with the highest c_2.
5. Choose a variable with the highest c_3.
6. Choose a variable with the highest c_*.
7. Further heuristics deal with the "basic blocks" of a proof — equalities of small types (`eqtABeq0`, `eqtXeqAB` (AND), `eqt11m2` (XOR)). A variable occurring in those more often is better to split on, since it generates more modifiers after substitution (for example, if we substitute $A = 0$ into `eqtXeqAB`, we get a modifier $X = 0$, and if we substitute $A = 1$ there, we get a modifier $X = B$, both being very desirable outcomes).

   Choose a variable with the highest number of occurrences in equalities of the type `eqtABeq0`.
8. Choose a variable with the highest number of occurrences in equalities of the type `eqtXeqAB` as the first variable.
9. Choose a variable with the highest number of occurrences in equalities of the type `eqtXeqAB` as one of the last two variables plus the number of occurrences in equalities of the type `eqt11m2`.
10. Choose a variable with the highest number of occurrences in equalities of the type `Special` with at most 3 variables plus the number of occurrences in equalities of the type `eqt124`.
11. Choose a variable with the highest number of occurrences in equalities of all the above-mentioned types (except for `eqt124`), where `Special` equalities with 2 variables are counted twice.
12. Choose a variable with the highest total number of occurrences.

Then the value to be examined first is determined based on the occurrences of the chosen variable. This highly heuristic part is better read in the solver's code (`src/general/variablestatistics.cc`).

3.5 Interfaces and command-line options

3.5.1 Command-line options

The command-line options of `basolver` are the following:

- `--equiv` (flag, default=off) — may be given only with two input files; with this flag the solver checks equivalence of the two input circuits: it produces a formula which is unsatisfiable if and only if the two input circuits are equivalent and solves it; if two files are given with this flag the solver merges the equalities stating that all corresponding inputs and outputs of two input circuits are equal and solves the resulting set
- `--booth` (flag, default=off) — use special rules for for Booth multipliers (under development, currently does not really help)
- `--html` (flag, default=off) — write html output
- `--nosets` (flag, default=off) — do not write most of the sets in html output (sets are printed only before splittings)

3.5.2 Interfaces

Below we describe the main interfaces of `basolver`. All of them are declared in the file `src/solve.h`. One can use this file together with the solver library, which can be made by typing
`make solverlib`
in the top-level directory.

- `int* solveCNF(const int* const* const array)`

  Solves an input CNF formula given as array of clauses (a null-terminated array of null-terminated arrays of literals). If the formula is satisfiable, returns a satisfying assignment (as a null-terminated array of literals). Otherwise, returns NULL.

- `int* solveCNF(const char* filename)`

  Reads a CNF formula given in DIMACS format from a given file and solves it. If the formula is satisfiable, returns a satisfying assignment (as a null-terminated array of literals). Otherwise, returns NULL.

- `bool checkEquivalence(const char* filename1, const char* filename2)`

  Reads either two input circuits or a circuit and a specification given in either ISCAS or RTL format and checks whether they are equivalent (creates a formula stating that on some input two input circuits have different outputs). Returns true iff they are equivalent (note however that they are equivalent iff the resulting formula is unsatisfiable!).

- `bool solve(const char* filename1, const char* filename2)`

  Reads two input circuits given in either ISCAS or RTL format, adds equalities stating that their inputs and outputs are equal, and solves the resulting formula. Returns true iff the resulting formula is satisfiable.

3.6 Tables for equality types

3.6.1 How equality types are transferred into each other under assignments

1. Type `eqtXeqAB`, $0 = -d + ab$:
   (a) under $d = 0$: $0 = 0 + ab$, type `eqtABeq0`, see table for `eqtABeq0`;
   (b) under $d = 1$: $a = 1$, $b = 1$ (all modifiers of the kind $x = 0$ or $x = y$ have type `Special`);
   (c) under $a = 0$: $d = 0$;
   (d) under $a = 1$: $d = b$;

2. Type `eqtXeqABC`, $0 = -d + abc$:
   (a) under $d = 0$: $0 = abc$, type `Special`;
   (b) under $d = 1$: $a = 1$, $b = 1$, $c = 1$;
   (c) under $c = 0$: $d = 0$;
   (d) under $c = 1$: $0 = -d + ab$, type `eqtXeqAB`, see table for `eqtXeqAB`;
3. Type `eqtXeqABpACpBCm2ABC`, $0 = -d + ab + ac + bc - 2abc$:
   (a) under $d = 0$: $0 = ab + ac + bc - 2abc$, type `Special`;
   (b) under $d = 1$: $0 = -1 + ab + ac + bc - 2abc$, type `Special`;
   (c) under $c = 0$: $0 = -d + ab$, type `eqtXeqAB`, see table for `eqtXeqAB`;
   (d) under $c = 1$: $0 = -d + a + b - ab$, type `eqtXeqAB`, see table for `eqtXeqAB`;
4. Type `eqt124`, $d = a + b + c - 2ab - 2ac - 2bc + 4abc$:
   (a) under $d = 0$: $a = b + c - 2bc$, type `eqt11m2`, see table for `eqt11m2`;
   (b) under $d = 1$: $a = 1 - b - c + 2bc$, type `eqt11m2`, see table for `eqt11m2`;
5. Type `eqtABeq0`, $0 = ab$:
   (a) under $a = 0$: $0 = 0$, a tautology;
   (b) under $a = 1$: $b = 0$;
6. Type `eqt11m2`, $d = a + b - 2ab$:
   (a) under $d = 0$: $a = b$;
   (b) under $d = 1$: $a = 1 - b$;
7. Type `eqtDeqABpACmABC`, $0 = -d + ab + ac - abc$:
   (a) under $d = 0$: $0 = ab + ac - abc$, type `Special`;
   (b) under $d = 1$: $a = 1$, $0 = -1 + b + c - bc$, type `Special`;
   (c) under $a = 0$: $d = 0$;
   (d) under $a = 1$: $0 = -d + b + c - bc$, type `eqtXeqAB`, see table for `eqtXeqAB`;
   (e) under $b = 0$: $0 = -d + ac$, type `eqtXeqAB`, see table for `eqtXeqAB`;
   (f) under $b = 1$: $d = a$;
8. Type `eqtYeqACpDmACD`, $0 = -d + ab + c - abc$:
   (a) under $d = 0$: $0 = ab + c - abc$, type `Special`;
   (b) under $d = 1$: $0 = -1 + ab + c - abc$, type `Special`;
   (c) under $a = 0$: $d = c$;
   (d) under $a = 1$: $0 = -d + b + c - bc$, type `eqtXeqAB`, see table for `eqtXeqAB`;
   (e) under $c = 0$: $0 = -d + ab$, type `eqtXeqAB`, see table for `eqtXeqAB`;
   (f) under $c = 1$: $d = c$;
9. Type `eqtYeqACpnAB`, $0 = -y + ac + (1 - a)b$.
   (a) $b = 0$: $0 = -y + ac$, type `eqtXeqAB` on literals y, a, c;
   (b) $a = 1$ (resp. $a = 0$): $y = c$ (resp. $y = b$);

(c) Other substitutions result in `Special` equalities.

10. Type `eqtWeqXmXYZ`: $0 = -w + x - xyz$:

    (a) $x = 0$: $w = 0$;

    (b) $y = 0$ or $z = 0$: $w = x$;

    (c) $y = 1$ (resp. $z = 1$): $0 = -w + x - xz$ (resp. $0 = -w + x - xy$), type `eqtXeqAB`;

    (d) Other substitutions result in `Special` equalities.

11. Type `Special` remains `Special` under all substitutions, unless it is recognized as one of the automatically recognized types `eqtXeqAB`, `eqt11m2`, or `eqtABeq0`.

3.6.2 Symmetries inside equality types

The records in this table show that an equality may not change if some of its literals change places with other literals. This should be taken into account in the rules. Besides, we show here the semantics of literal numbers in different type (what numbers literals have — that is, `var1` is the variable one would get if one asked for `getVar1()`, `var2` — for `getVar2()` and so on). We show only "pure" types that have literals (such types as `Special` or `BoothSubResult` would be meaningless here).

1. Type `eqtXeqAB` is symmetrical under exchanging A and B; `var1` $= X$, `var2` $= A$, `var3` $= B$.

2. Type `eqtABeq0` is symmetrical under exchanging A and B; `var1` $= A$, `var2` $= B$.

3. Type `eqt11m1` ($0 = -x + a + b - ab$) is symmetrical under exchanging a and b; `var1` $= x$, `var2` $= a$, `var3` $= b$.

4. Type `eqt11m2` ($0 = -x + a + b - 2ab$) is symmetrical under exchanging all three variables. Besides, only the overall parity of literal signs matters (all positive or two negative signs would result in the same equality). `var1` $= x$, `var2` $= a$, `var3` $= b$.

5. Type `eqt124` ($0 = -x + a + b + c - 2ab - 2ac - 2bc + 4abc$) is symmetrical under permuting of all four variables. Besides, only the overall parity of literal signs matters. `var1` $= x$, `var2` $= a$, `var3` $= b$, `var4` $= c$.

6. Type `eqtXeqABC` is symmetrical under permuting A, B and C; `var1` $= X$, `var2` $= A$, `var3` $= B$, `var4` $= C$.

7. Type `eqtXeqABpACpBCm2ABC` is symmetrical under permuting A, B and C; `var1` $= X$, `var2` $= A$, `var3` $= B$, `var4` $= C$.

8. Type `eqtDeqABpACmABC` is symmetrical under exchanging B and C; `var1` $= D$, `var2` $= A$, `var3` $= B$, `var4` $= C$.

9. Type `eqtYeqACpDmACD` is symmetrical under exchanging A and C; `var1` $= Y$, `var2` $= A$, `var3` $= C$, `var4` $= D$.

10. Type `eqtYeqACpnAB` is symmetrical under exchanging B and C (note that the sign of A will change altogether); `var1` $= Y$, `var2` $= A$, `var3` $= B$, `var4` $= C$.

11. Type `eqtWeqXmXYZ` is symmetrical under exchanging Y and Z; `var1` $= W$, `var2` $= X$, `var3` $= Y$, `var4` $= Z$.

12. Type `eqtZeqWXpVXm2VWX` is symmetrical under exchanging W and V; `var1` $= Z$, `var2` $= X$, `var3` $= V$, `var4` $= W$.

13. Type `eqtSeqXpVnWm2XVnW` is symmetrical under exchanging S and X; `var1` $= S$, `var2` $= X$, `var3` $= V$, `var4` $= W$.

14. Type `eqtCeqWXYpnWnXY` is symmetrical under exchanging W and X; `var1` $= C$, `var2` $= W$, `var3` $= X$, `var4` $= Y$.

15. Type `eqtCeqXZmXYpnWYnZp2WXYnZ` is not symmetrical at all; `var1` $= C$, `var2` $= W$, `var3` $= X$, `var4` $= Y$, `var5` $= Z$.

16. Type `eqtCeqYnWYxorZ` is not symmetrical at all; `var1` $= C$, `var2` $= W$, `var3` $= Y$, `var4` $= Z$.

17. Type `eqtVeqYZLS` is symmetrical under permuting Y, Z, L, S; `var1` $= V$, `var2` $= Y$, `var3` $= Z$, `var4` $= L$, `var5` $= S$.

18. Type `eqtVeqYZnotLS` is symmetrical under exchanging Y and Z, and also L and S; `var1` $= V$, `var2` $= Y$, `var3` $= Z$, `var4` $= L$, `var5` $= S$.

Chapter 4

The idea behind the `basolver` proofs

4.1 Introduction

The idea behind our proofs is *propositional inductive inference*, i.e., proving formulas of propositional logic by gradually maintaining some invariant instead of using the induction (which is very hard to automatize). For example, one could prove that $32+31+\ldots+1=\frac{32\cdot(32+1)}{2}$ by gradually proving $1=\frac{1\cdot(1+1)}{2}$, $2+1=\frac{2\cdot(2+1)}{2}$, $\ldots$, instead of proving the quantified formula $n+\ldots+1=\frac{n\cdot(n+1)}{2}$ and instantiating $n=32$.

The solver we have created is designed to demonstrate this idea by proving and disproving equivalences between boolean circuits (specifically multipliers). To achieve scalability, we have undertaken the search for polynomial-sized proofs of equivalences between different circuits. After a "manual" proof had been found, we tried to reformulate the proof as a sequence of applications of relatively simple rules, which we would implement in the solver.

Thus, we were facing two challenges. First was to devise the proofs themselves, and the second was to cut the proofs into rule applications in an appropriate way. On one hand, the rules should be simple and straightforward. On the other hand, they should not be applicable too often, thus immersing the solver into an endless sea of redundant rule applications. These requirements were usually contradictory. However, we have successfully retained the balance — on many pairs of multipliers (but, unfortunately, not on the Booth multipliers — see Sect. 4.6).

In this chapter we describe the manual proofs and the way they follow from our set of rules. We will also describe the boolean CNF instances which correspond to different equivalence checking problems.

All these formulas have similar structure. Each consists of two descriptions of circuits whose equivalence is to be proven. Besides, it also describes the fact that the outputs of these two circuits should be equal. The latter part of the formula is very similar for different instances.

The basic idea behind all proofs is to think of some invariant (this thinking is of a creative nature and, unfortunately, has to be left to human beings) and carry the invariant on inductively through all the steps (levels of multipliers) up to the outputs, where it would deliver its *coup de grace* in proving the equivalence. Let us make an example of this general framework — a most useful example, in fact.

One of the basic invariant that may be observed throughout all proofs presented in this section deals with the sums of carry bits on corresponding levels of different multipliers (by a level we mean here roughly the set of full adders implementing the summation of the result of the previous level with the next element of the sum, which is usually the second input multiplied by the corresponding bit of the first input). Namely, the sum of carry bits on a given level of one multiplier should be equal to the sum of carry bits on the same level of any other multiplier. Note that this holds for all proofs, not only equivalence proofs of identical multipliers.

Multipliers we consider contain full adders. A general technique that we use throughout all proofs is to convert the boolean description of a full adder into the linear equality of the kind $x+y+z-sum-2\cdot carry=0$, where x, y, and z are inputs of the full adder, sum is the result, and $carry$ is the carry bit.

One last remark. Many rules (including the rules necessary for the theoretical proof) may generate equalities that are not important for the theoretical proof; thus, one should not attempt to find some theoretical basis for each of the created equalities: some are just there because the rule could not be restricted with 100% efficiency. One should rather follow the derivation in the theoretical proof, finding corresponding equalities in the actual inference.

4.2 Checking two add-stepper multipliers for equivalence

In this section we provide the first and simplest polynomial-sized proof of equivalence of two circuits. We concentrate on two simplest circuits — add-stepper multipliers (see Sect. A.1) — and prove that they are equivalent. This proof opens the series of polynomial-sized proofs of equivalences between different circuits that our proof system was designed for. Being the simplest one of all proofs, the proof in this subsection makes no sense by itself: it checks the equivalence of two identical circuits. However, it provides a simple (perhaps, the simplest possible) yet important example to illustrate the ideas that will be further employed in all other proofs.

Note that `basolver` is able to prove equivalence not only of two add-stepper multipliers, but of any two identical multipliers (at least, among those described in this document). The proofs all follow the same lines, so we describe only this one in details.

Let us consider how our algorithm works on the formula encoding the equivalence of two add-stepper multipliers. Such a formula has the following structure:

- the first part of the clauses encodes a simple add-stepper multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N+1$ bits and $Y = \{y_0, \ldots, y_N\}$ having $N+1$ bits (see Sect. A.1);
- the second part of the clauses encodes the very same multiplier with different variables — namely, we take the first part and change index 1 to 2;
- the third part of the clauses encodes the fact that some two outputs are not equal (thus, the circuits are equivalent iff the formula is unsatisfiable); to achieve that, this part introduces *miter*, which is equal to 1; for every pair of outputs $\{r_{ij1}$, where $i = 0$ or $j = N$, $t_{NN1}\}$, and $\{r_{ij2}$, where $i = 0$ or $j = N$, $t_{NN2}\}$ (for the first and the second circuits respectively) we add clauses that look like ($0 \le i + j \le 2N + 1$):

$$\left.\begin{aligned} \neg r_{ij1} \vee r_{ij2} \vee xout_{i+j} \\ r_{ij1} \vee \neg r_{ij2} \vee xout_{i+j} \\ r_{ij1} \vee r_{ij2} \vee \neg xout_{i+j} \\ \neg r_{ij1} \vee \neg r_{ij2} \vee \neg xout_{i+j} \end{aligned}\right\} \quad xout_{i+j} = r_{ij1} \oplus r_{ij2}$$

$$\left.\begin{aligned} \neg t_{NN1} \vee t_{NN2} \vee xout_{N+N+1} \\ t_{NN1} \vee \neg t_{NN2} \vee xout_{N+N+1} \\ t_{NN1} \vee t_{NN2} \vee \neg xout_{N+N+1} \\ \neg t_{NN1} \vee \neg t_{NN2} \vee \neg xout_{N+N+1} \end{aligned}\right\} \quad xout_{N+N+1} = t_{NN1} \oplus t_{NN2}$$

$$\left.\begin{aligned} \neg xout_0 \vee miter \\ \vdots \\ \neg xout_{N+N+1} \vee miter \\ \neg miter \vee xout_0 \vee \ldots \vee xout_{N+N+1} \end{aligned}\right\} \quad miter = xout_0 \vee \ldots \vee xout_{N+N+1}$$

$$miter\} \qquad miter = 1$$

In this proof and further we add the subscript 1 to variables occurring in the first part of the formula (variables that concern the first multiplier), and the subscript 2 to the variables occurring in the second part of the formula.

To prove this formula we need the following rules:

- `CircuitFormTranslationRule`;
- `GatesEquivalenceRule`.

Let us follow, step by step, the algorithm's run on this formula. In each of the following sections, we shall thus illustrate how we get a polynomial-sized proof. The solver should not split on any variables during this run: everything is done by the proof system rules.

1. A modifier $miter = 1$ will be recognized by `TransferRule`.
2. By the `CircuitFormTranslationRule` all clauses, except for clauses containing $miter$, will be transformed into functional encodings of the gates (equalities written to the right of the clauses in the CNF descriptions of multipliers).
3. The equality $miter = 1$ will be recognized as a modifier; equalities generated by `CircuitFormTranslationRule` will be recognized as clauses.
4. Because of the $miter = 1$ equality, the following equalities will be deleted:

   $$\neg xout_0 \vee miter,\ 0 \leq i \leq 2N+1, \qquad miter$$

   from the `Clauses` set and the following clause:

   $$\neg miter \vee xout_0 \vee \ldots \vee xout_{N+N+1}$$

   will be substituted for

   $$xout_0 \vee \ldots \vee xout_{N+N+1};$$

5. The clause $xout_0 \vee \ldots \vee xout_{N+N+1}$ will be recognized as a clause;
6. The rule `GatesEquivalenceRule` generates the following objects that will be recognized as modifiers:

   $$c_{ij1} = c_{ij2},\ i = 0..N,\ j = 1..N \text{ and } r_{i01} = r_{i02},\ i = 0..N$$

   using only equalities containing the bits of factors x_i and y_j, $0 \leq i \leq N$, $0 \leq j \leq N$;
7. We prove the equality of outputs of cells with numbers (i, j) in different circuits: $r_{ij1} = r_{ij2}$ and $t_{ij1} = t_{ij2}$. For the equality proof it is necessary to have equalities of the following two inputs of a cell with number (i, j): $r_{i+1,j-1,1} = r_{i+1,j-1,2}$ and $t_{i-1,j,1} = t_{i-1,j,2}$ (or, for some cells, one of these inputs).

   After that, by `GatesEquivalenceRule`, we derive from the following equalities:

   $$\begin{aligned}
   k_{ij1} &= c_{ij2} + r_{i+1,j-1,2} - 2c_{ij2}r_{i+1,j-1,2}, && (4.2.1)\\
   k_{ij2} &= c_{ij2} + r_{i+1,j-1,2} - 2c_{ij2}r_{i+1,j-1,2}, && (4.2.2)\\
   l_{ij1} &= c_{ij2}r_{i+1,j-1,2}, && (4.2.3)\\
   l_{ij2} &= c_{ij2}r_{i+1,j-1,2}, && (4.2.4)\\
   m_{ij1} &= t_{i-1,j,2}r_{i+1,j-1,2}, && (4.2.5)\\
   m_{ij2} &= t_{i-1,j,2}r_{i+1,j-1,2}, && (4.2.6)\\
   n_{ij1} &= t_{i-1,j,2}c_{ij2}, && (4.2.7)\\
   n_{ij2} &= t_{i-1,j,2}c_{ij2}, && (4.2.8)
   \end{aligned}$$

the equalities $k_{ij1} = k_{ij2}$, $l_{ij1} = l_{ij2}$, $m_{ij1} = m_{ij2}$, $n_{ij1} = n_{ij2}$. When clauses containing variables k_{ij1}, l_{ij1}, m_{ij1}, n_{ij1}, simplified by the above-mentioned equalities by modifiers, appear in the `Clauses` set, the `GatesEquivalenceRule` will apply to them again, creating equalities $r_{ij1} = r_{ij2}$ and $e_{ij1} = e_{ij2}$.

After that, by `GatesEquivalenceRule`, we get the last necessary equality $t_{ij1} = t_{ij2}$ from the equalities $t_{ij1} = e_{ij2} + n_{ij2} - e_{ij2}n_{ij2}$ and $t_{ij2} = e_{ij2} + n_{ij2} - e_{ij2}n_{ij2}$.

8. After the equality $r_{ij1} = r_{ij2}$, where $i = 0$ or $j = N$, appear in the `Modifiers` set, it will be substituted by to the equality

$$xout_J = r_{ij1} + r_{ij2} - 2r_{ij1}r_{ij,2},$$

where $i = 0$ or $j = N$, $i + j = J$. As a result of the substitution, we get the equality $xout_J = 0$ that will afterwards be substituted to the clause

$$xout_J \vee \ldots \vee xout_{N+N+1},$$

simplifying it up to

$$xout_{J+1} \vee \ldots \vee xout_{N+N+1}.$$

9. At the end of the proof, the final equality

$$xout_{N+N+1} = 0$$

will be substituted into the unit clause

$$xout_{N+N+1},$$

thus arriving at the contradiction.

4.3 Checking a diagonal multiplier against an add-stepper multiplier

In this section we provide the proof in the proof system of our solver that checks the equivalence of two multipliers: the simplest add-stepper multiplier (it usually serves us as a test circuit against which we test all other circuits) and the diagonal multiplier (see Fig. A.1 for both of the circuits). This equivalence translates into the following CNF formula:

- the first part of the clauses encodes a simple add-stepper multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N + 1$ bits and $Y = \{y_0, \ldots, y_N\}$ having $N + 1$ bits (see Sect. A.1);
- the second part of the clauses encodes a simple diagonal multiplier (see Sect. A.2) with the same inputs.
- the third part of the clauses encodes the fact that some two outputs are not equal (thus, the circuits are equivalent iff the formula is unsatisfiable); to achieve that, this part introduces *miter*, which is equal to 1; for every pair of outputs $\{r_{ij1}$, where $i = 0$ or $j = N$, $t_{NN1}\}$, and $\{r_{ij2}$, where $i = 0$ or $j = N$, $t_{NN2}\}$ (for the first and the second circuits respectively) we add clauses that look like ($0 \le i + j \le 2N + 1$):

$$\left.\begin{aligned} \neg r_{0j1} \vee r_{0j2} \vee xout_j \\ r_{0j1} \vee \neg r_{0j2} \vee xout_j \\ r_{0j1} \vee r_{0j2} \vee \neg xout_j \\ \neg r_{0j1} \vee \neg r_{0j2} \vee \neg xout_j \end{aligned}\right\} \quad xout_{i+j} = r_{ij1} \oplus r_{ij2}$$

$$\left.\begin{aligned} \neg r_{i+1N1} \vee r_{iN+1,2} \vee xout_{i+N+1} \\ r_{i+1N1} \vee \neg r_{iN+1,2} \vee xout_{i+N+1} \\ r_{i+1N1} \vee r_{iN+1,2} \vee \neg xout_{i+N+1} \\ \neg r_{i+1N1} \vee \neg r_{iN+1,2} \vee \neg xout_{i+N+1} \end{aligned}\right\} \quad xout_{i+N+1} = r_{i+1N1} \oplus r_{iN+1,2}$$

$$\left.\begin{array}{r}\neg t_{NN1} \vee t_{N-1N+1,2} \vee xout_{N+N+1}\\ t_{NN1} \vee \neg t_{N-1N+1,2} \vee xout_{N+N+1}\\ t_{NN1} \vee t_{N-1N+1,2} \vee \neg xout_{N+N+1}\\ \neg t_{NN1} \vee \neg t_{N-1N+1,2} \vee \neg xout_{N+N+1}\end{array}\right\} \quad xout_{N+N+1} = t_{NN1} \oplus t_{N-1N+1,2}$$

$$\left.\begin{array}{r}\neg xout_0 \vee miter\\ \vdots\\ \neg xout_{N+N+1} \vee miter\\ \neg miter \vee xout_0 \vee \ldots \vee xout_{N+N+1}\end{array}\right\} \quad miter = xout_0 \vee \ldots \vee xout_{N+N+1}$$

$$miter\} \qquad miter = 1$$

To prove this formula we need the following rules:

- `CircuitFormTranslationRule`;
- `OR3GenerationRule`;
- `XOR3GenerationRule`;
- `Linearization3Rule`;
- `Linearization2Rule`;
- `SummationRule`;
- `GatesEquivalenceRule`;

Let us follow the algorithm through the proof:

1. A modifier $miter = 1$ will be recognized by `TransferRule`.
2. By `CircuitFormTranslationRule` all clauses, except for clauses containing $miter$, will be transformed into functional encodings of the gates (equalities written to the right of the clauses in the CNF descriptions of multipliers).
3. The equality $miter = 1$ will be recognized as a modifier; equalities generated by `CircuitFormTranslationRule` will be recognized as clauses.
4. Because of the $miter = 1$ equality, the following clauses will be deleted:

   $$\neg xout_0 \vee miter,\ 0 \leq i \leq 2N+1, \qquad miter$$

   from the `Clauses` set and the following clause:

   $$\neg miter \vee xout_0 \vee \ldots \vee xout_{N+N+1}$$

   will be substituted for

   $$xout_0 \vee \ldots \vee xout_{N+N+1};$$

5. For each full adder (in both circuits) with three inputs[1], `OR3GenerationRule`, `XOR3GenerationRule`, and `Linearization3Rule` apply one after another and generate the following equalities:

   $$0 = -r_{ij} + c_{ij} + r + t - 2t_{ij}, \tag{4.3.1}$$

   where r and t are inputs of the $\{ij\}$-th cell containing a full adder.

[1] `Linearization2Rule` takes care of full adders with two inputs.

The actual inference goes as follows[2]:

$$0 = -k_{ij} + c_{ij} + r - 2c_{ij}r \quad \text{(original equality)} \tag{4.3.2}$$
$$0 = -r_{ij} + k_{ij} + t - 2k_{ij}t \quad \text{(original equality)} \tag{4.3.3}$$
$$0 = -l_{ij} + c_{ij}r \quad \text{(original equality)} \tag{4.3.4}$$
$$0 = -m_{ij} + c_{ij}t \quad \text{(original equality)} \tag{4.3.5}$$
$$0 = -n_{ij} + rt \quad \text{(original equality)} \tag{4.3.6}$$
$$0 = -e_{ij} + l_{ij} + m_{ij} - l_{ij}m_{ij} \quad \text{(original equality)} \tag{4.3.7}$$
$$0 = -t_{ij} + e_{ij} + n_{ij} - e_{ij}n_{ij} \quad \text{(original equality)} \tag{4.3.8}$$
$$0 = -r_{ij} + c_{ij} + t + r - 2c_{ij}t - 2c_{ij}r - 2rt + 4c_{ij}rt \quad \text{(from (4.3.2), (4.3.3) by } \texttt{XOR3GenerationRule}\text{)} \tag{4.3.9}$$
$$0 = -e_{ij} + c_{ij}r + m_{ij} - c_{ij}rm_{ij} \quad \text{(from (4.3.4), (4.3.7) by } \texttt{OR3GenerationRule}\text{)} \tag{4.3.10}$$
$$0 = -e_{ij} + c_{ij}r + c_{ij}t - c_{ij}rt \quad \text{(from (4.3.5), (4.3.10) by } \texttt{OR3GenerationRule}\text{)} \tag{4.3.11}$$
$$0 = -t_{ij} + e_{ij} + rt - e_{ij}rt \quad \text{(from (4.3.6), (4.3.8) by } \texttt{OR3GenerationRule}\text{)} \tag{4.3.12}$$
$$0 = -t_{ij} + c_{ij}t + c_{ij}r + rt - 2c_{ij}rt \quad \text{(from (4.3.11), (4.3.12) by } \texttt{OR3GenerationRule}\text{)} \tag{4.3.13}$$
$$0 = -r_{ij} + c_{ij} + t + r - 2t_{ij} \quad \text{(from (4.3.9), (4.3.13) by } \texttt{Linearization3Rule}\text{)} \tag{4.3.14}$$

6. The clause $xout_0 \vee \ldots \vee xout_{N+N+1}$ and equalities (4.3.1) are recognized as clauses.

7. `Linearization2Rule` generates an alternative representation for the full adders with two inputs:

$$0 = -r_{ij} + c_{ij} + s - 2t_{ij}. \tag{4.3.15}$$

8. `GatesEquivalenceRule` generates the equalities $0 = -c_{ij1} + c_{ij2},\ i = 0..N,\ j = 1..N$, $0 = -r_{i01} + r_{i02},\ i = 0..N$ using one-equality clauses containing x_i and y_j, $0 \le i \le N$, $0 \le j \le N$.

9. Equalities generated by `GatesEquivalenceRule` are recognized as modifiers; equalities generated by other rules are recognized as clauses.

10. Let us show how the invariants described in Sect. 4.1 apply to this case. The explanation given may with necessary modifications be applied to every proof we have designed. We denote by tr_k the sum of the carry bits t_{ij1} on the k-th level of the first circuit ($i+j=k$); by TR_k we denote the sum of the carry bits t_{ij2} on the k-th level of the second circuit ($i+j=k$). From the previous level (by the induction hypothesis, so to say) we have $tr_k = TR_k$. All equalities of the form (4.3.1) and (4.3.15) (note that in the two latter equalities variables from $0 = -tr_k + TR_k$ may occur as t or s) for the k-th level of each of the circuits are summed up by `SummationRule` with the equalities $0 = -xout_k + p_k + P_k - 2p_kP_k$ (or $0 = xout_k - p_k + P_k - 2xout_kP_k$, depending on the variable ordering) and $0 = -tr_k + TR_k$, thus generating

$$0 = -xout_k + 2(p_k - p_kP_k + tr_{k+1} - TR_{k+1}). \tag{4.3.16}$$

From the latter equality `StrongNormalizeRule` generates a modifier $xout_k = 0$ and

$$0 = p_k - p_kP_k + tr_{k+1} - TR_{k+1}. \tag{4.3.17}$$

[2]equalities $0 = -k_{ij} + c_{ij} + r - 2c_{ij}r$, $0 = -r_{ij} + k_{ij} + t - 2k_{ij}t$ and $0 = -r_{ij} + c_{ij} + t + r - 2c_{ij}t - 2c_{ij}r - 2rt + 4c_{ij}rt$ may have different normal form — it depends on the order of the variables.

Then, the modifier $xout_k = 0$ is substituted into $0 = -xout_k + p_k + P_k - 2p_kP_k$ (or into $0 = xout_k - p_k + P_k - 2xout_kP_k$, depending on the variable ordering), generating modifier $0 = -p_k + P_k$. The latter modifier is substituted into (4.3.17) and takes part in obtaining the equality between carry bit sums $0 = -tr_{k+1} + TR_{k+1}$ for the current level.

11. On the last iteration the following equality:

$$xout_{N+N+1} = 0$$

will be substituted into the unit clause

$$xout_{N+N+1},$$

thus arriving at the contradiction.

4.4 Checking a modified add-stepper multiplier against an add-stepper multiplier

We provide here the handmade proof of this equivalence for two reasons. First, the proof illustrates the basic principles of our techniques, and, second, the proof will be used as a component in the handmade proof of the equivalence between simplified Booth and add-stepper multipliers. The equivalence of the two circuits in question translates into the following CNF formula:

- the first part of the clauses encodes a modified add-stepper multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N + 1$ bits and $Y = \{y_0, \ldots, y_N\}$ having $N + 1$ bits (see Sect. A.3);
- the second part of the clauses encodes a simple add-stepper multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N + 1$ bits and $Y = \{y_0, \ldots, y_N\}$ having $N + 1$ bits (see Sect. A.1);
- the third part of the clauses encodes the fact that some two outputs are not equal (thus, the circuits are equivalent iff the formula is unsatisfiable); to achieve that, this part introduces *miter*, which is equal to 1;

$$\left.\begin{array}{r} \neg p_{j1} \vee p_{j2} \vee xout_j \\ p_{j1} \vee \neg p_{j2} \vee xout_j \\ p_{j1} \vee p_{j2} \vee \neg xout_j \\ \neg p_{j1} \vee \neg p_{j2} \vee \neg xout_j \end{array}\right\} \quad xout_j = p_{j1} \oplus p_{j2}$$

$$\left.\begin{array}{r} \neg xout_0 \vee miter \\ \vdots \\ \neg xout_{2N+1} \vee miter \\ \neg miter \vee xout_0 \vee \ldots \vee xout_{2N+1} \end{array}\right\} \quad miter = xout_0 \vee \ldots \vee xout_{2N+1}$$

$$miter\} \qquad miter = 1$$

To prove this formula we need the following rules:

- `CircuitFormTranslationRule`;
- `OR3GenerationRule`;
- `OR3GenerationRule2`;
- `XOR3GenerationRule`;

- `BackLinearization2Rule`;
- `Linearization3Rule`;
- `Linearization2Rule`;
- `SummationRule`;
- `XorSummationRule2`;
- `SummationRule2`;
- `GatesEquivalenceRule`;

To prove the equivalence of the two curcuits we show that our rules produce the following equation for each full adder of each circuit:

$$r = r' + t' + c - 2t \ ,$$

where r and t are right and top outputs of a full adder, r', t' and c are its inputs, and also r' and t' are outputs of some other full adders. We have already shown that this equality is generated for each full adder of an add-stepper multiplier (it simply states that sum of inputs of a full adder is equal to the sum of its right output and its top output multiplied by 2). Below we show that this equality is also generated for a modified add-stepper multiplier.

Initially, we are given the following equalities:

$$\begin{aligned} k &= x + r' - 2xr' && (4.4.1)\\ o &= k + s' - 2kt' && (4.4.2)\\ l &= xr' && (4.4.3)\\ m &= xs' && (4.4.4)\\ n &= r's' && (4.4.5)\\ e &= l + m - lm && (4.4.6)\\ s &= n + e - ne && (4.4.7)\\ b &= oy && (4.4.8)\\ d &= r' - yr' && (4.4.9)\\ r &= d + b - db && (4.4.10) \end{aligned}$$

As in the proof of equivalence of add-stepper and diagonal multipliers, the rules produce the following equality:

$$o = x + r' + s' - 2s \ . \qquad (4.4.11)$$

Then, the inference goes as follows:

$r = r' + oy - yr'$ (from (4.4.8), (4.4.9), (4.4.10) by `OR3GenerationRule` and `OR3GenerationRule2`) (4.4.12)

$r = xy + r' + s'y - 2sy$ (from (4.4.12), (4.4.11) by `SummationIIRule`) (4.4.13)

$r = c + r' + s'y - 2sy$ (from (4.4.13), $c = xy$ by `BackT2SubstitutionRule`) (4.4.14)

By induction hypothesis, we already have an equality $t' = s'y$. From this equality and (4.4.14) `BackT2SubstitutionRule` generates the following equality:

$$r = c + r' + t' - 2sy \ . \qquad (4.4.15)$$

Finally, `GatesEquivalenceRule` and `Summation` generate $t = sy$ (which we will use on the next level) and prove the equivalence of the right outputs of corresponding full adders in the two circuits.

By doing this, we will prove the equivalence of right outputs of all pairs of corresponding full adders in the two circuits. It immediately implies the equivalence of the two circuits.

4.5 Checking a simplified Wallace multiplier against an add-stepper multiplier

By "simplified Wallace" we mean a multiplier with additions each making two numbers out of three numbers in logarithmic time in parallel, these additions are organized in a logarithmic-depth tree — see Sect. A.4 for details.

Just like we have done in the previous sections, we encode the equivalence problem into CNF as follows:

- the first part of the clauses encodes a simplified Wallace multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N+1$ bits and $Y = \{y_0, \ldots, y_N\}$ having $N+1$ bits (see Sect. A.4);
- the second part of the clauses encodes a simple add-stepper multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N+1$ bits and $Y = \{y_0, \ldots, y_N\}$ having $N+1$ bits (see Sect. A.1);
- the third part of the clauses encodes the fact that some two outputs are not equal (thus, the circuits are equivalent iff the formula is unsatisfiable); to achieve that, this part introduces *miter*, which is equal to 1; for every similar pair of outputs p_{i*} of the two circuits we add the following clauses ($0 \leq j \leq 2N+1$):

$$\left.\begin{array}{r} \neg p_{j1} \vee p_{j2} \vee xout_j \\ p_{j1} \vee \neg p_{j2} \vee xout_j \\ p_{j1} \vee p_{j2} \vee \neg xout_j \\ \neg p_{j1} \vee \neg p_{j2} \vee \neg xout_j \end{array}\right\} \quad xout_j = p_{j1} \oplus p_{j2}$$

$$\left.\begin{array}{r} \neg xout_0 \vee miter \\ \vdots \\ \neg xout_{2N+1} \vee miter \\ \neg miter \vee xout_0 \vee \ldots \vee xout_{2N+1} \end{array}\right\} \quad miter = xout_0 \vee \ldots \vee xout_{2N+1}$$

$$miter\} \qquad miter = 1$$

We shall prove that our proof system suffices for this equivalence proof by showing that this proof has the very same structure as the diagonal against add-stepper proof (described in Sect. 4.3). Suppose that the multiplier has obtained as input $X = \{x_0, \ldots, x_n\}$ and $Y = \{y_0, \ldots, y_n\}$. Let us define as the i-th level of the circuit the set of full adders adding the products of input bits $x_k y_l$, $k + l = i$, and carry bits from level $i-1$. A parallel simplified Wallace multiplier adds up the same numbers, obtained by bitwise multiplication, as a simple add-stepper multiplier. The only difference is in the summation order. Thus, the i-th level of the simplified Wallace multiplier adds up the same numbers as the i-th level of the simple add-stepper multiplier. Therefore, the i-th output of the circuit is a XOR of all inputs on the i-th level, and the sum of the carry bits fn the i-th level is equal to the sum of the carry bits of the add-stepper multiplier. In other words, the multiplication process satisfies the same invariants that hold for the add-stepper against diagonal proof. Therefore, the algorithm proving equivalence of add-stepper and diagonal multipliers also proves equivalence of the pair in question (simplified Wallace multiplier versus add-stepper multiplier). This claim is supported by practice: as soon as our solver had enough rules to solve add step against diagonal without splitting (by the power of the proof system only), it was immediately able to solve simplified Wallace and add-stepper for equivalence.

4.6 Checking a simplified Booth multiplier against an add-stepper multiplier

In this section we consider how our algorithm is supposed to work on the formula that consists of two parts (it really does work on 2-by-2 instances, but not further — see Sect. 4.7 for details).

- The first part of the clauses describes a simplified Booth multiplier (see Sect. A.5) with inputs $X = \{x_0, \ldots, x_N\}$ ($N+1$ bits) and $Y = \{y_0, \ldots, y_N\}$ ($N+1$ bits). This formula was described in the previous section, and we keep that notation, only adding 1 to every index to distinguish between two parts of the formula.
- The second part of the clauses describes a basic add-step multiplier with inputs $X = \{x_0, \ldots, x_N\}$ ($N+1$ bits) and $Y = \{y_0, \ldots, y_N\}$ ($N+1$ bits). We keep the notation of Sect. A.1, adding 2 to every index.
- The third part of the clauses encodes *miter* which is the `OR` of variables corresponding to the `XOR`'s of results of the two circuits. The formula is satisfiable iff there is a bug, so we set *miter* to 1. Thus, for every pair of similar outputs p_{i*} we add the following clauses ($0 \leq j \leq 2N+1$):

$$\left.\begin{array}{r} \neg p_{j1} \vee p_{j2} \vee xout_j \\ p_{j1} \vee \neg p_{j2} \vee xout_j \\ p_{j1} \vee p_{j2} \vee \neg xout_j \\ \neg p_{j1} \vee \neg p_{j2} \vee \neg xout_j \end{array}\right\} \quad xout_j = p_{j1} \oplus p_{j2}$$

$$\left.\begin{array}{r} \neg xout_0 \vee miter \\ \vdots \\ \neg xout_{2N+1} \vee miter \\ \neg miter \vee xout_0 \vee \ldots \vee xout_{2N+1} \end{array}\right\} \quad miter = xout_0 \vee \ldots \vee xout_{2N+1}$$

$$miter\} \qquad miter = 1$$

To prove this formula we need the following rules:

- `CircuitFormTranslationRule`;
- `OR3GenerationRule`;
- `OR3GenerationRule2`;
- `XOR3GenerationRule`;
- `ORXOR3Rule`;
- `XOROR2Part1Rule`;
- `XOROR2Part2Rule`;
- `XOROR2Part3Rule`;
- `ORORRule`;
- `BackT2SubstitutionRule`;
- `BoothSubRule`;
- `Linearization3Rule`;
- `Linearization2Rule`;
- `SummationRule`;
- `GatesEquivalenceRule`;

- `StrongNormalizeRule`;

In the simplified Booth multiplier the following numbers are summed up like in the usual add-stepper multiplier: $(\sum_{i=0}^{N+1} c_{ij}2^i)\cdot 2^j$, $0 \le j \le N+1$, where

$$\begin{aligned}
c_{i0} &= -y_0x_i + y_0(1 - w_{i-1} + 2x_iw_{i-1}),\\
c_{ij} &= y_{j-1}x_i - y_jx_i + y_j(1-y_{j-1})(1 - w_{i-1} + 2x_iw_{i-1}), \qquad 1 \le j \le N,\\
c_{i,N+1} &= y_Nx_i,\\
c_{N+1,0} &= y_j(1-w_N) + c_{N+1j-1} - 2y_j(1-w_N)c_{N+1j-1},\\
c_{N+1,j} &= y_j(1-y_{j-1})(1-w_N) + c_{N+1j-1} - 2y_j(1-y_{j-1})(1-w_N)c_{N+1j-1},
\end{aligned}$$

which result from the following initial clauses:

$$\begin{aligned}
s_j &= y_j + y_{j-1} - 2y_jy_{j-1}, && (4.6.1)\\
v_{ij} &= y_j + x_i - 2y_jx_i, && (4.6.2)\\
u_{i-1j} &= w_{i-1}y_j, && (4.6.3)\\
g_{ij} &= v_{ij} + u_{i-1j} - 2v_{ij}u_{i-1j}, && (4.6.4)\\
c_{ij} &= g_{ij}s_j && (4.6.5)
\end{aligned}$$

by the `XOR3GenerationRule` rule:

$$g_{ij} = y_j + x_i + u_{i-1j} - 2y_jx_i - 2x_iu_{i-1j} - 2y_ju_{i-1j} + 4y_jx_iu_{i-1j} \qquad (4.6.2) \text{ and } (4.6.4), \quad (4.6.6)$$

`ORXOR3Rule`:

$$g_{ij} = y_j + x_i - w_{i-1}y_j - 2y_jx_i + 2x_iw_{i-1}y_j \qquad (4.6.3) \text{ and } (4.6.6), \quad (4.6.7)$$

and `XOROR2Rule` (we also note the part of the rule which is used):

$$\begin{aligned}
c_{ij} &= g_{ij}y_j + g_{ij}y_{j-1} - 2g_{ij}y_jy_{j-1} && (4.6.5) \text{ and } (4.6.1), \text{ part 1},\\
c_{ij} &= y_{j-1}x_i - y_jx_i + y_j(1-y_{j-1})(1 - w_{i-1} + 2x_iw_{i-1}) && (4.6) \text{ and } (4.6.7), \text{ part 3}.
\end{aligned}$$

Similar to the above reasoning we get c_{ij} for the special cases $i = N+1$, $j = [0, N+1]$.

In turn, the add-stepper multiplier adds up numbers $(\sum_{i=0}^{N} C_{ij}2^i)\cdot 2^j$, $0 \le j \le N$, where

$$C_{ij} = x_iy_j$$

Just like in the proof of equivalence between add-stepper and diagonal multipliers, for both circuits we derive by `SummationRule` that the output of the circuit on a given level[3] is equal to the sum c_{ij} of carry bits from the previous level minus the doubled sum of carry bits from this level:

$$\sum_{j=0}^{l} c_{j,l-j} + tr_{l-1} - 2tr_l = p_l. \qquad (4.6.8)$$

By `BoothSubRule`, all c_{ij} are substituted[4] into (4.6.8), and we thus, by `BoothSubRule`, derive for the simplified Booth multiplier[5] the following:

$$\begin{aligned}
&-y_0x_l + y_0(1 - w_{l-1} + 2x_lw_{l-1}) +\\
&\quad + \sum_{j=1}^{l}(y_{j-1}x_{l-j} - y_jx_{l-j} + y_j(1-y_{j-1})(1 - w_{l-j-1} + 2x_{l-j}w_{l-j-1}))) + tr_{l-1} - 2tr_l = p_l, \qquad (4.6.9)
\end{aligned}$$

[3] We prove here that outputs are equal for all levels $l \le N$; for levels with $l > N$ the proof is given further.

[4] The substitutions are carried out one by one, starting from the leftmost linear sum (with the cell c_{ij} with maximal i) on a given level; see the description of `BoothSubRule`.

[5] Only for levels $l \le N$; for $l > N$, the representation will be a little different, since we will need to consider bits representing the sign of the input number.

where p_l is the lth output of the multiplier, tr_{l-1} is the sum of carry bits from the previous level, and tr_l is the sum of carry bits of the current level.

By the induction hypothesis,

$$tr_{l-1} - y_0(1 - w_{l-1}) + y_0 x_{l-1} + \sum_{j=1}^{l-1}(y_j x_{l-1-j} - (1 - y_{j-1})y_j(1 - w_{l-1-j})) = Tr_{l-1},$$

where Tr_{l-1} is the sum of the carry bits from the previous level in the add-stepper multiplier. Let us substitute it into (4.6.9):

$$\begin{aligned} - y_0 x_l + y_0(1 - w_{l-1} + 2x_l w_{l-1}) + \\ + \sum_{j=1}^{l}(y_{j-1}x_{l-j} - y_j x_{l-j} + y_j(1 - y_{j-1})(1 - w_{l-j-1} + 2x_{l-j}w_{l-j-1}))) + \\ + (Tr_{l-1} + y_0(1 - w_{l-1}) - y_0 x_{l-1} + \sum_{j=1}^{l-1}((1 - y_{j-1})y_j(1 - w_{l-1-j}) - y_j x_{l-1-j})) - \\ - 2tr_l = p_l. \end{aligned} \quad (4.6.10)$$

We note that

$$1 - w_{i-1} + 2x_i w_{i-1} + (1 - w_{i-1}) = 2(1 - w_i), \qquad 2x_0 = 2(1 - w_0) \quad (4.6.11)$$

and rewrite the equality (4.6.10) (eliminating $\sum_{j=1}^{l} y_{j-1}x_{l-j} - \sum_{j=0}^{l-1} y_j x_{l-1-j}$ which is zero):

$$-\sum_{j=0}^{l} y_j x_{l-j} + Tr_{l-1} - 2(tr_l - y_0(1 - w_l) - \sum_{j=1}^{l}(1 - y_{j-1})y_j(1 - w_{l-j})) = p_l. \quad (4.6.12)$$

Let us rewrite (4.6.12) in such a way that every $y_j x_{l-j}$ would occur there positively[6]:

$$\sum_{j=0}^{l} y_j x_{l-j} + Tr_{l-1} - 2(tr_l - y_0(1 - w_l) + y_0 x_l - \sum_{j=1}^{l}((1 - y_{j-1})y_j(1 - w_{l-j}) - y_j x_{l-j})) = p_l. \quad (4.6.13)$$

For the add-stepper multiplier on level l we derive (by summing up with `SummationRule` from the linear equalities of the alternative representation of the full adder)

$$\sum_{j=0}^{l} C_{j,l-j} + Tr_{l-1} - 2Tr_l = P_l, \quad (4.6.14)$$

where P_l is the lth output of the multiplier, Tr_{l-1} is the sum of carry bits from the previous level, and Tr_l is the sum of carry bits from the current level.

We add (4.6.13) and (4.6.14) (by `BoothSubRule`) and apply `BackT2SubstitutionRule` (that allows to substitute c_{ij} from the add-stepper multiplier, (4.6.14), instead of $x_i y_j$ that were in the simplified Booth, (4.6.13)), obtaining $p_l + P_l = 2S$. Together with $xout_l = p_l \oplus P_l$ this gives $xout_l = 0$, $p_l = 1 - P_l$. Substituting the latter in the sum (4.6.13) and (4.6.14) generated the following equalities necessary for the next level of proof: $tr_l - y_0(1 - w_l) + y_0 x_l + \sum_{j=1}^{l}(y_j x_{l-j} - (1 - y_{j-1})y_j(1 - w_{l-j})) = Tr_l$.

We now turn to consider levels $l \geq n + 1$. On these levels additional summands are needed to represent negative numbers:

$$c_{ij} = y_j(1 - y_{j-1})(1 - w_n), \qquad i \geq n + 1.$$

[6] by `BackT2SubstitutionRule` instead of $y_j x_{l-j}$ every equality generated by `BoothSubRule` will contain $C_{j,l-j}$ from the add-stepper multiplier

Since with $i \geq n+1$ it is true that $c_{ij} = c_{i+1j}$, in fact we add only one extra summand, which is equal to the sum of carry bits (modulo 2) of the complementary code from the previous columns. The sum obtained by our rules (obtained just like the sum (4.6.9), by applying `BoothSubRule` to the linear sum):

$$\sum_{j=0}^{l-n} y_j(1-y_{j-1})(1-w_n)+ \\ + \sum_{j=l-n+1}^{n} (y_{j-1}x_{l-j} - y_j x_{l-j} + y_j(1-y_{j-1})(1-w_{l-j-1}+ \\ + 2x_{l-j}w_{l-j-1}))) + tr_{l-1} - 2tr_l = p_l, \quad (4.6.15)$$

where p_l is the lth output of the multiplier, tr_{l-1} is the sum of the carry bits from the previous level, and tr_l is the sum of the carry bits from the current level. The inductive hypothesis looks like the following:

$$tr_{l-1} + \sum_{j=0}^{l-n-1} (y_j x_n - y_j(1-w_n)) + \sum_{j=l-n}^{n} (y_j x_{l-1-j} - (1-y_{j-1})y_j(1-w_{l-1-j})) = Tr_{l-1}.$$

Further proof is similar to the proof carried out on levels $l \leq n$. We substitute the induction hypothesis into (4.6.15), annihilate similar summands, and simplify it with (4.6.11). The resulting equality for the simplified Booth multiplier couples with the equality of the same level for the add-stepper multiplier, thus deriving that the outputs of two multipliers are equal and providing the induction hypothesis for the next level.

4.7 Problems with the Booth and simplified Booth multipliers

As we have already mentioned, we first tried to "teach" the solver to solve for the equivalence between simplified Booth and add-stepper multipliers. The automated proof was supposed to proceed along the lines described in the previous section.

However, during the implementation we encountered a serious problem which we could not overcome. The problem is that the proof provided has degree four (it extensively uses polynomials of degree four), and in order to derive the contradiction these polynomials should be matching against each other. However, there are too many of them, so as a result of the summation we obtain a vast amount of excessive, unnecessary equalities (thousands and tens of thousands on very simple examples). This does not allow the solver to finish in reasonable time.

To reach equilibrium between taking too long and not solving at all, we had to impose certain restrictions on some of the generation rules in order for them to generate a reasonable number of objects. The restrictions, unfortunately, may interfere with the theoretical proof, especially when different permutations of the variables (and, consequently, different variable orderings) are given as input.

We made several attempts to subtly tune the `Summation` rule, so that it would sum up what is needed and discard what is not needed. Unfortunately, these situations are very hard to distinguish (if at all possible without knowing the history of the premises), so we made very little progress, succeeding only in proving the equivalence of simplified Booth and add-stepper multipliers on two inputs for a given (natural) variable ordering. On all bigger benchmarks the solver was overwhelmed by the number of clauses generated by `BoothSubRule` (a rule designed specifically for the Booth multipliers). We failed to weaken this rule so that it would generate a reasonable amount of objects while keeping its ability to generate the automated proof. We even had to give up solvability on all permutations of the variables (there exist permutations where the theoretical proof will not be generated by the solver) because the restrictions we might impose keeping all permutations solvable would not permit us to solve even 2-by-2 multipliers for equivalence in reasonable time (the rules would generate too many objects).

An interesting direction for further research would be to lower the degree of polynomials used in the proof. If it were possible, we would derive the final result much easier. Unfortunately, we were not able to

lower the degree theoretically. It is possible that with more effort the degree problem can be overcome — the problem was finally solved for Booth multipliers on two inputs. However, a nontrivial research is needed here.

Chapter 5

Empirical data

As we said in Sect. 4.1, specific rules of our solver were designed for the verification of specific classes of benchmarks. In this chapter we give the empirical results attained at such benchmarks and compare them to the behavior of state-of-the-art SAT solvers[1]. For most our experiments we used a linux machine with a 1 GHz Intel Pentium-III processor (actually, the machine had two such processors, but this is not important, because our solver does not use parallel processes). The random seeds used for generating multipliers on this CPU are equal to the number of bits in each input. For our massive experiments with "tweaked" benchmarks we used a 1.8 GHz Intel Pentium-IV machine. The random seeds used for generating multipliers on this CPU are taken at random, the corresponding tables are sorted by CPU time.

The notation "X Y i" for a benchmark means "equivalence checking of i-bits multiplier X against multiplier Y". The acronyms for multipliers are:

a — add-steper,

b — Booth,

d — diagonal,

ma — modified add-stepper,

p — parallel (simplified Wallace),

B2X — multiplier X with a bug for one pair of inputs (see Sect. 5.3).

The CPU time in given in seconds, this is the "user time" reported by the `bash` built-in command `time`. For `basolver` we also give the number of generated objects and also the number of nodes if DPLL has been used. For other solvers we give the number of nodes (for zchaff, minisat, and SatELite this is the figure reported by the solver as the number of decisions). The acronym "DNS" means "did not start": we usually did not launch solvers on larger instances if they failed smaller similar instances.

5.1 Equivalence checking of (almost) identical circuits

At first glance, the problem of checking two identical circuits for equivalence is trivial, especially if one is given two circuits with the same order of vertices so that there is no need even to solve the isomorphism problem of the two directed acyclic graphs. However, if such problem is formulated as a boolean formula

[1] We used the following solvers:
`berkmin`, version 561, `http://eigold.tripod.com/BerkMin.html` ;
`minisat`, version 1.13, `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html` ;
`zchaff`, version 2004.11.15, `http://www.princeton.edu/~chaff/zchaff.html` ;
`SatELite`, version 1.0, `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/SatELite.html` .

benchmark	basolver		satelite		minisat		zchaff		berkmin	
	time	objects	time	nodes	time	nodes	time	nodes	time	nodes
a a 3	0	249	0	115	0	122	0	87	0	105
a a 4	0	524	0	518	0	428	0	376	0	367
a a 5	0	891	0	4293	0	2838	0	1810	0	1736
a a 6	1	1335	0	16322	0	13495	3	8787	0	8001
a a 7	2	1877	2	48171	4	44265	44	48738	7	27799
a a 8	3	2523	13	198660	28	155867	170	151489	78	103758
a a 9	4	3243	60	593067	111	406378	891	503765	736	381589
a a 10	7	4067	623	2939097	851	1872638	>3600	-	>3600	-
a a 11	10	4977	>3600	-	>3600	-	>3600	-	>3600	-
a a 12	14	5994	>3600	-	>3600	-	>3600	-	>3600	-
a a 13	19	7091	DNS	-	DNS	-	DNS	-	DNS	-
a a 14	27	8271	DNS	-	DNS	-	DNS	-	DNS	-
a a 15	36	9552	DNS	-	DNS	-	DNS	-	DNS	-
a a 16	48	10919	DNS	-	DNS	-	DNS	-	DNS	-
a a 32	931	45387	DNS	-	DNS	-	DNS	-	DNS	-
a a 44	3384	86688	DNS	-	DNS	-	DNS	-	DNS	-
b b 3	0	894	0	103	0	105	0	125	0	147
b b 4	1	1449	0	717	0	633	0	798	0	736
b b 5	2	2226	0	3232	0	2185	0	2924	0	3560
b b 6	3	3162	1	14356	2	12110	5	12963	2	11029
b b 7	5	4279	5	59318	21	70340	94	94158	34	49556
b b 8	9	5554	33	243138	111	217784	1610	586797	427	243854
b b 9	13	7034	310	1164194	1083	1292327	>3600	-	3298	943271
b b 10	20	8566	1913	4721773	>3600	-	>3600	-	>3600	-
b b 11	29	10246	>3600	-	>3600	-	>3600	-	>3600	-
b b 12	42	12393	>3600	-	>3600	-	>3600	-	>3600	-
b b 13	59	14315	DNS	-	DNS	-	DNS	-	DNS	-
b b 14	81	16533	DNS	-	DNS	-	DNS	-	DNS	-
b b 15	111	19001	DNS	-	DNS	-	DNS	-	DNS	-
b b 16	154	21318	DNS	-	DNS	-	DNS	-	DNS	-
b b 32	2181	83653	DNS	-	DNS	-	DNS	-	DNS	-
b b 36	3482	105880	DNS	-	DNS	-	DNS	-	DNS	-

Table 5.1: Equivalence checking of identical multipliers

in conjunctive normal form (by replacing each gate with corresponding 2..4 boolean clauses and stating that an OR of XORs of the pairs of outputs is equal to 1 — the so-called *xor miter*), it becomes hard for contemporary SAT solvers. However, solving such CNFs does not make any problem to our solver, because `basolver` is able to extract the circuits from boolean formulas (see Table 5.1). Note that all formulas considered in our experiments were *reshuffled* similarly to how it was done at SAT Competitions[2], i.e., the order of variables and clauses and the signs of literals are taken at random.

Going slightly further, consider the same problem, but when one of the circuits has been slightly "tweaked", i.e., a randomly selected gate is replaced by a small circuit computing the same boolean function; in our experiments we used the following circuits (note that we do not have special rules for dealing with

[2] `www.satcompetition.org`

them):

$$\begin{aligned}
x \oplus y &= (x \vee y) \wedge \neg(x \wedge y),\\
x \oplus y &= y \oplus (x \vee x),\\
x \oplus y &= \neg x \oplus \neg y,\\
x \vee y &= \neg(x \wedge y),\\
x \vee y &= y \vee (x \wedge y),\\
x \vee y &= (x \wedge y) \vee (x \oplus y),\\
x \wedge y &= \neg(x \vee y),\\
x \wedge y &= y \wedge (x \vee x),\\
x \wedge y &= (x \vee y) \wedge \neg(x \oplus y),\\
\neg x &= \neg(x \wedge x),\\
\neg x &= \neg(x \vee x),\\
\neg x &= x \oplus (x \vee \neg x),\\
\mathrm{id}(x) &= x \wedge x,\\
\mathrm{id}(x) &= x \vee x,\\
\mathrm{id}(x) &= \neg\neg x.
\end{aligned}$$

Of course, some of these "tweaks" appeared to be trivial and the corresponding benchmarks were solved by `basolver` without splitting. Others required splitting, but in most cases the formulas have been successfully solved, see Tables 5.2–5.3 for massive experiments with 12-bits multipliers (we do not provide any data for other solvers, because none of them was able to solve benchmarks of this size in 2 minutes, which was the time limit; actually, we believe that none of the solvers is able to solve them in an hour).

5.2 Equivalence of multipliers based on different algorithms

The main purpose of our solver was checking the equivalence of multipliers based on different algorithms. While other contemporary SAT solvers are unable to solve such benchmarks for 12-bits circuits in less than one hour, our solver was able to solve the problems for 32-bits multipliers, see Table 5.4.

Similarly to Sect. 5.1, if one of the circuits is slightly "tweaked", our solver is frequently still able to solve the equivalence checking problem (though the success rate is lower; it is a challenge to devise a good splitting heuristic that would improve this behavior), see Tables 5.5–5.8.

5.3 Finding rare bugs

In the experiments above we were *proving* circuit equivalence, i.e., showing that the corresponding boolean formulas are unsatisfiable. What if one of the circuits in the above examples has a bug? If one introduces a "random" bug-like flipping the value of a gate, a circuit always certainly becomes completely unusable, and the resulting boolean formula will have a lot of satisfying assignments. Therefore, such formulas are trivial for contemporary SAT solvers.

However, if a bug appears rarely, the situation becomes quite different. (Unfortunately, it is difficult to generate such buggy circuits automatically for testing.) We experimented with bugs that appear only for one pair of inputs.The experimental data presented in Table 5.9 demonstrates that our solver is much better in finding such rare bugs than other SAT solvers.

benchmark	basolver			benchmark	basolver			benchmark	basolver		
	time	nodes	objects		time	nodes	objects		time	nodes	objects
a a 12 (1)	6	1	5989	a a 12 (2)	7	1	5972	a a 12 (3)	7	1	5975
a a 12 (4)	7	1	5975	a a 12 (5)	7	1	5975	a a 12 (6)	7	1	5978
a a 12 (7)	7	1	5978	a a 12 (8)	7	1	5978	a a 12 (9)	7	1	5979
a a 12 (10)	7	1	5981	a a 12 (11)	7	1	5981	a a 12 (12)	7	1	5981
a a 12 (13)	7	1	5982	a a 12 (14)	7	1	5982	a a 12 (15)	7	1	5983
a a 12 (16)	7	1	5983	a a 12 (17)	7	1	5983	a a 12 (18)	7	1	5984
a a 12 (19)	7	1	5984	a a 12 (20)	7	1	5984	a a 12 (21)	7	1	5984
a a 12 (22)	7	1	5984	a a 12 (23)	7	1	5985	a a 12 (24)	7	1	5986
a a 12 (25)	7	1	5987	a a 12 (26)	7	1	5987	a a 12 (27)	7	1	5987
a a 12 (28)	7	1	5988	a a 12 (29)	7	1	5988	a a 12 (30)	7	1	5988
a a 12 (31)	7	1	5988	a a 12 (32)	7	1	5989	a a 12 (33)	7	1	5990
a a 12 (34)	7	1	5990	a a 12 (35)	7	1	5990	a a 12 (36)	7	1	5990
a a 12 (37)	7	1	5990	a a 12 (38)	7	1	5990	a a 12 (39)	7	1	5991
a a 12 (40)	7	1	5991	a a 12 (41)	7	1	5992	a a 12 (42)	7	1	5992
a a 12 (43)	7	1	5993	a a 12 (44)	7	1	5993	a a 12 (45)	7	1	5993
a a 12 (46)	7	1	5993	a a 12 (47)	7	1	5994	a a 12 (48)	7	1	5995
a a 12 (49)	7	1	5995	a a 12 (50)	7	1	5996	a a 12 (51)	7	1	5997
a a 12 (52)	7	1	5997	a a 12 (53)	7	1	6000	a a 12 (54)	7	1	6001
a a 12 (55)	7	1	6002	a a 12 (56)	7	1	6002	a a 12 (57)	7	1	6002
a a 12 (58)	7	1	6002	a a 12 (59)	7	1	6003	a a 12 (60)	7	1	6004
a a 12 (61)	7	1	6005	a a 12 (62)	7	1	6005	a a 12 (63)	7	1	6008
a a 12 (64)	7	1	6021	a a 12 (65)	7	1	6022	a a 12 (66)	7	1	6025
a a 12 (67)	7	1	6027	a a 12 (68)	7	1	7155	a a 12 (69)	7	1	7158
a a 12 (70)	8	1	7118	a a 12 (71)	8	1	7442	a a 12 (72)	8	1	8507
a a 12 (73)	8	1	8751	a a 12 (74)	8	3	7645	a a 12 (75)	8	3	7935
a a 12 (76)	9	1	7626	a a 12 (77)	9	3	8945	a a 12 (78)	9	3	9544
a a 12 (79)	9	3	9732	a a 12 (80)	9	3	10273	a a 12 (81)	9	3	15916
a a 12 (82)	10	1	9973	a a 12 (83)	10	3	10077	a a 12 (84)	11	3	11615
a a 12 (85)	11	3	11878	a a 12 (86)	11	3	12167	a a 12 (87)	12	3	12424
a a 12 (88)	12	3	13332	a a 12 (89)	12	3	13564	a a 12 (90)	13	3	14777
a a 12 (91)	13	3	15790	a a 12 (92)	14	3	16189	a a 12 (93)	14	3	16247
a a 12 (94)	14	3	16893	a a 12 (95)	15	3	18510	a a 12 (96)	15	3	18687
a a 12 (97)	16	3	19320	a a 12 (98)	16	3	19945	a a 12 (99)	16	3	20223
a a 12 (100)	16	3	20681								

Table 5.2: Equivalence checking of *almost* identical add-steper multipliers

benchmark	basolver			benchmark	basolver			benchmark	basolver		
	time	nodes	objects		time	nodes	objects		time	nodes	objects
b b 12 (1)	23	1	12162	b b 12 (2)	24	1	12196	b b 12 (3)	24	1	12222
b b 12 (4)	24	1	12319	b b 12 (5)	24	1	12343	b b 12 (6)	24	1	12399
b b 12 (7)	25	1	11863	b b 12 (8)	25	1	11943	b b 12 (9)	25	1	11981
b b 12 (10)	25	1	12009	b b 12 (11)	25	1	12022	b b 12 (12)	25	1	12023
b b 12 (13)	25	1	12048	b b 12 (14)	25	1	12051	b b 12 (15)	25	1	12062
b b 12 (16)	25	1	12071	b b 12 (17)	25	1	12080	b b 12 (18)	25	1	12085
b b 12 (19)	25	1	12102	b b 12 (20)	25	1	12112	b b 12 (21)	25	1	12118
b b 12 (22)	25	1	12119	b b 12 (23)	25	1	12123	b b 12 (24)	25	1	12131
b b 12 (25)	25	1	12145	b b 12 (26)	25	1	12161	b b 12 (27)	25	1	12172
b b 12 (28)	25	1	12175	b b 12 (29)	25	1	12196	b b 12 (30)	25	1	12205
b b 12 (31)	25	1	12208	b b 12 (32)	25	1	12208	b b 12 (33)	25	1	12211
b b 12 (34)	25	1	12215	b b 12 (35)	25	1	12221	b b 12 (36)	25	1	12223
b b 12 (37)	25	1	12230	b b 12 (38)	25	1	12234	b b 12 (39)	25	1	12236
b b 12 (40)	25	1	12239	b b 12 (41)	25	1	12250	b b 12 (42)	25	1	12258
b b 12 (43)	25	1	12287	b b 12 (44)	25	1	12295	b b 12 (45)	25	1	12300
b b 12 (46)	25	1	12308	b b 12 (47)	25	1	12333	b b 12 (48)	25	1	12343
b b 12 (49)	25	1	12370	b b 12 (50)	25	1	12564	b b 12 (51)	26	1	12074
b b 12 (52)	26	1	12108	b b 12 (53)	26	1	12166	b b 12 (54)	26	1	12172
b b 12 (55)	26	1	12177	b b 12 (56)	26	1	12205	b b 12 (57)	26	1	12227
b b 12 (58)	26	1	12233	b b 12 (59)	26	1	12246	b b 12 (60)	27	1	12069
b b 12 (61)	27	1	12077	b b 12 (62)	27	1	12140	b b 12 (63)	27	1	12199
b b 12 (64)	27	1	12250	b b 12 (65)	27	1	12364	b b 12 (66)	27	1	12394
b b 12 (67)	28	1	12184	b b 12 (68)	28	1	12314	b b 12 (69)	29	1	12037
b b 12 (70)	41	1	18841	b b 12 (71)	44	1	20311	b b 12 (72)	44	1	23787
b b 12 (73)	45	1	24982	b b 12 (74)	45	1	26093	b b 12 (75)	46	1	19413
b b 12 (76)	46	1	21981	b b 12 (77)	48	1	22479	b b 12 (78)	50	1	23028
b b 12 (79)	53	3	23361	b b 12 (80)	53	3	23492	b b 12 (81)	57	3	28098
b b 12 (82)	58	3	27590	b b 12 (83)	60	3	22024	b b 12 (84)	63	3	28622
b b 12 (85)	71	3	31655	b b 12 (86)	74	3	24746	b b 12 (87)	75	5	56066
b b 12 (88)	76	3	27324	b b 12 (89)	85	3	42655	b b 12 (90)	85	3	48057
b b 12 (91)	91	3	28441	b b 12 (92)	102	3	32076	b b 12 (93)	>120	-	-
b b 12 (94)	>120	-	-	b b 12 (95)	>120	-	-	b b 12 (96)	>120	-	-
b b 12 (97)	>120	-	-	b b 12 (98)	>120	-	-	b b 12 (99)	>120	-	-
b b 12 (100)	>120	-	-								

Table 5.3: Equivalence checking of *almost* identical Booth multipliers

benchmark	basolver		satelite		minisat		zchaff		berkmin	
	time	objects	time	nodes	time	nodes	time	nodes	time	nodes
a d 3	0	336	0	124	0	104	0	97	0	82
a d 4	0	686	0	761	0	460	0	366	0	337
a d 5	1	1096	0	3319	0	2454	0	2016	0	1804
a d 6	3	1608	0	17700	0	13357	3	8440	0	7298
a d 7	5	2269	3	58653	4	49418	55	55901	9	31865
a d 8	9	3045	36	408307	34	220624	2703	729221	111	122219
a d 9	14	3896	180	1311700	257	859193	>3600	-	914	436860
a d 10	21	4844	1217	4857347	1904	3592679	>3600	-	>3600	-
a d 11	33	5961	>3600	-	>3600	-	>3600	-	>3600	-
a d 12	45	7051	>3600	-	>3600	-	>3600	-	>3600	-
a d 13	59	8291	DNS	-	DNS	-	DNS	-	DNS	-
a d 14	83	9630	DNS	-	DNS	-	DNS	-	DNS	-
a d 15	108	11054	DNS	-	DNS	-	DNS	-	DNS	-
a d 16	142	12679	DNS	-	DNS	-	DNS	-	DNS	-
a d 32	2910	52117	DNS	-	DNS	-	DNS	-	DNS	-
a ma 3	0	461	0	91	0	80	0	75	0	99
a ma 4	0	974	0	464	0	445	0	486	0	394
a ma 5	2	1741	0	2459	0	2090	0	2139	0	1861
a ma 6	3	2739	0	13022	0	8271	2	8501	0	8112
a ma 7	6	3889	2	43059	3	36937	32	43307	9	27634
a ma 8	10	5577	13	213300	22	114900	644	286185	141	138612
a ma 9	14	7483	79	752019	207	523838	>3600	-	826	422824
a ma 10	20	9291	317	1974776	957	1762697	>3600	-	>3600	-
a ma 11	30	12123	2680	8018089	>3600	-	>3600	-	>3600	-
a ma 12	41	14490	>3600	-	>3600	-	>3600	-	>3600	-
a ma 13	57	17613	DNS	-	DNS	-	DNS	-	DNS	-
a ma 14	73	20569	DNS	-	DNS	-	DNS	-	DNS	-
a ma 15	99	24746	DNS	-	DNS	-	DNS	-	DNS	-
a ma 16	138	29847	DNS	-	DNS	-	DNS	-	DNS	-
a ma 32	2413	167659	DNS	-	DNS	-	DNS	-	DNS	-
a p 3	0	324	0	83	0	82	0	73	0	96
a p 4	0	682	0	558	0	452	0	490	0	389
a p 5	1	1145	0	2930	0	3048	0	1969	0	1910
a p 6	3	1676	0	16327	1	14837	3	8927	1	9202
a p 7	5	2380	4	74181	5	53974	67	62006	10	32187
a p 8	9	3150	35	377711	68	331299	>3600	-	153	140722
a p 9	15	3963	281	1587935	440	1305423	>3600	-	1817	623680
a p 10	23	5062	1671	6148344	>3600	-	>3600	-	>3600	-
a p 11	37	6158	>3600	-	>3600	-	>3600	-	>3600	-
a p 12	49	7324	>3600	-	>3600	-	>3600	-	>3600	-
a p 13	66	8649	DNS	-	DNS	-	DNS	-	DNS	-
a p 14	90	10011	DNS	-	DNS	-	DNS	-	DNS	-
a p 15	119	11546	DNS	-	DNS	-	DNS	-	DNS	-
a p 16	158	13134	DNS	-	DNS	-	DNS	-	DNS	-
a p 32	3116	53357	DNS	-	DNS	-	DNS	-	DNS	-
d p 3	0	320	0	140	0	61	0	83	0	89
d p 4	0	687	0	423	0	360	0	469	0	327
d p 5	1	1136	0	3950	0	2438	0	1850	0	1655
d p 6	3	1833	0	15181	1	13712	3	8625	0	7375
d p 7	5	2464	4	73889	4	45919	51	54623	12	34944
d p 8	8	3276	30	335246	40	233217	2383	634325	143	131290
d p 9	13	4158	296	1744046	737	1994614	>3600	-	1798	630135
d p 10	20	5192	2203	7261321	2991	5182824	>3600	-	>3600	-
d p 11	32	6460	>3600	-	>3600	-	>3600	-	>3600	-
d p 12	42	7574	>3600	-	>3600	-	>3600	-	>3600	-
d p 13	56	8863	DNS	-	DNS	-	DNS	-	DNS	-
d p 14	76	10227	DNS	-	DNS	-	DNS	-	DNS	-
d p 15	106	12013	DNS	-	DNS	-	DNS	-	DNS	-
d p 16	134	13472	DNS	-	DNS	-	DNS	-	DNS	-
d p 32	2605	55089	DNS	-	DNS	-	DNS	-	DNS	-

Table 5.4: Equivalence checking of pairs of different multipliers

benchmark	basolver			benchmark	basolver			benchmark	basolver		
	time	nodes	objects		time	nodes	objects		time	nodes	objects
a d 12 (1)	22	1	7011	a d 12 (2)	25	1	7053	a d 12 (3)	26	1	7026
a d 12 (4)	26	1	7029	a d 12 (5)	26	1	7035	a d 12 (6)	26	1	7040
a d 12 (7)	26	1	7061	a d 12 (8)	26	1	7068	a d 12 (9)	26	1	7077
a d 12 (10)	26	1	7096	a d 12 (11)	26	1	7153	a d 12 (12)	27	1	6991
a d 12 (13)	27	1	7002	a d 12 (14)	27	1	7002	a d 12 (15)	27	1	7009
a d 12 (16)	27	1	7015	a d 12 (17)	27	1	7019	a d 12 (18)	27	1	7022
a d 12 (19)	27	1	7026	a d 12 (20)	27	1	7029	a d 12 (21)	27	1	7030
a d 12 (22)	27	1	7030	a d 12 (23)	27	1	7035	a d 12 (24)	27	1	7038
a d 12 (25)	27	1	7047	a d 12 (26)	27	1	7058	a d 12 (27)	27	1	7068
a d 12 (28)	27	1	7107	a d 12 (29)	27	1	7110	a d 12 (30)	28	1	6988
a d 12 (31)	28	1	7001	a d 12 (32)	28	1	7008	a d 12 (33)	28	1	7016
a d 12 (34)	28	1	7019	a d 12 (35)	28	1	7019	a d 12 (36)	28	1	7019
a d 12 (37)	28	1	7036	a d 12 (38)	28	1	7049	a d 12 (39)	28	1	7061
a d 12 (40)	28	1	7062	a d 12 (41)	28	1	7063	a d 12 (42)	28	1	7064
a d 12 (43)	28	1	7070	a d 12 (44)	28	1	7071	a d 12 (45)	28	1	7077
a d 12 (46)	28	1	7081	a d 12 (47)	28	1	7081	a d 12 (48)	28	1	7086
a d 12 (49)	28	1	7093	a d 12 (50)	28	1	7112	a d 12 (51)	28	1	7115
a d 12 (52)	28	1	7151	a d 12 (53)	29	1	6976	a d 12 (54)	29	1	6981
a d 12 (55)	29	1	7014	a d 12 (56)	29	1	7018	a d 12 (57)	29	1	7064
a d 12 (58)	29	1	7078	a d 12 (59)	29	1	7079	a d 12 (60)	29	1	7087
a d 12 (61)	29	1	7091	a d 12 (62)	29	1	7093	a d 12 (63)	29	1	7094
a d 12 (64)	29	1	7109	a d 12 (65)	29	1	7120	a d 12 (66)	29	1	7135
a d 12 (67)	29	1	7139	a d 12 (68)	29	1	7184	a d 12 (69)	30	1	7041
a d 12 (70)	30	1	7106	a d 12 (71)	30	1	7143	a d 12 (72)	30	1	7147
a d 12 (73)	30	1	7170	a d 12 (74)	31	3	7522	a d 12 (75)	32	1	7246
a d 12 (76)	32	3	7618	a d 12 (77)	34	3	7687	a d 12 (78)	34	3	8766
a d 12 (79)	37	3	8828	a d 12 (80)	39	3	8855	a d 12 (81)	44	3	9112
a d 12 (82)	45	3	9289	a d 12 (83)	46	3	8108	a d 12 (84)	>120	-	-
a d 12 (85)	>120	-	-	a d 12 (86)	>120	-	-	a d 12 (87)	>120	-	-
a d 12 (88)	>120	-	-	a d 12 (89)	>120	-	-	a d 12 (90)	>120	-	-
a d 12 (91)	>120	-	-	a d 12 (92)	>120	-	-	a d 12 (93)	>120	-	-
a d 12 (94)	>120	-	-	a d 12 (95)	>120	-	-	a d 12 (96)	>120	-	-
a d 12 (97)	>120	-	-	a d 12 (98)	>120	-	-	a d 12 (99)	>120	-	-
a d 12 (100)	>120	-	-								

Table 5.5: Equivalence checking of add-stepper multiplier against diagonal multiplier, with small random "tweak" to the former one

benchmark	basolver			benchmark	basolver			benchmark	basolver		
	time	nodes	objects		time	nodes	objects		time	nodes	objects
a ma 12 (1)	26	1	14194	a ma 12 (2)	26	1	14422	a ma 12 (3)	27	1	14110
a ma 12 (4)	27	1	14187	a ma 12 (5)	27	1	14252	a ma 12 (6)	27	1	14353
a ma 12 (7)	27	1	14378	a ma 12 (8)	27	1	14489	a ma 12 (9)	27	1	14514
a ma 12 (10)	28	1	14146	a ma 12 (11)	28	1	14162	a ma 12 (12)	28	1	14194
a ma 12 (13)	28	1	14281	a ma 12 (14)	28	1	14310	a ma 12 (15)	28	1	14342
a ma 12 (16)	28	1	14375	a ma 12 (17)	28	1	14385	a ma 12 (18)	28	1	14403
a ma 12 (19)	28	1	14420	a ma 12 (20)	28	1	14482	a ma 12 (21)	28	1	14520
a ma 12 (22)	28	1	14530	a ma 12 (23)	28	1	14554	a ma 12 (24)	28	1	14559
a ma 12 (25)	28	1	14559	a ma 12 (26)	28	1	14602	a ma 12 (27)	28	1	14606
a ma 12 (28)	28	1	14620	a ma 12 (29)	28	1	14623	a ma 12 (30)	28	1	14643
a ma 12 (31)	28	1	14685	a ma 12 (32)	28	1	14708	a ma 12 (33)	28	1	14761
a ma 12 (34)	28	1	14794	a ma 12 (35)	28	1	14832	a ma 12 (36)	28	1	14863
a ma 12 (37)	29	1	14441	a ma 12 (38)	29	1	14507	a ma 12 (39)	29	1	14559
a ma 12 (40)	29	1	14583	a ma 12 (41)	29	1	14598	a ma 12 (42)	29	1	14631
a ma 12 (43)	29	1	14654	a ma 12 (44)	29	1	14671	a ma 12 (45)	29	1	14724
a ma 12 (46)	29	1	14758	a ma 12 (47)	29	1	14762	a ma 12 (48)	29	1	14767
a ma 12 (49)	29	1	14780	a ma 12 (50)	29	1	14786	a ma 12 (51)	29	1	14791
a ma 12 (52)	29	1	14825	a ma 12 (53)	29	1	14843	a ma 12 (54)	29	1	14878
a ma 12 (55)	29	1	14896	a ma 12 (56)	29	1	14905	a ma 12 (57)	29	1	14909
a ma 12 (58)	29	1	14931	a ma 12 (59)	29	1	14934	a ma 12 (60)	29	1	14942
a ma 12 (61)	29	1	14991	a ma 12 (62)	29	1	15006	a ma 12 (63)	29	1	15059
a ma 12 (64)	30	1	14345	a ma 12 (65)	30	1	14516	a ma 12 (66)	30	1	14597
a ma 12 (67)	30	1	14729	a ma 12 (68)	30	1	14746	a ma 12 (69)	30	1	14783
a ma 12 (70)	30	1	14867	a ma 12 (71)	30	1	14921	a ma 12 (72)	30	1	15239
a ma 12 (73)	30	1	15409	a ma 12 (74)	57	3	28146	a ma 12 (75)	>120	-	-
a ma 12 (76)	>120	-	-	a ma 12 (77)	>120	-	-	a ma 12 (78)	>120	-	-
a ma 12 (79)	>120	-	-	a ma 12 (80)	>120	-	-	a ma 12 (81)	>120	-	-
a ma 12 (82)	>120	-	-	a ma 12 (83)	>120	-	-	a ma 12 (84)	>120	-	-
a ma 12 (85)	>120	-	-	a ma 12 (86)	>120	-	-	a ma 12 (87)	>120	-	-
a ma 12 (88)	>120	-	-	a ma 12 (89)	>120	-	-	a ma 12 (90)	>120	-	-
a ma 12 (91)	>120	-	-	a ma 12 (92)	>120	-	-	a ma 12 (93)	>120	-	-
a ma 12 (94)	>120	-	-	a ma 12 (95)	>120	-	-	a ma 12 (96)	>120	-	-
a ma 12 (97)	>120	-	-	a ma 12 (98)	>120	-	-	a ma 12 (99)	>120	-	-
a ma 12 (100)	>120	-	-								

Table 5.6: Equivalence checking of add-stepper multiplier against modified add-stepper multiplier, with small random "tweak" to the former one

benchmark	basolver			benchmark	basolver			benchmark	basolver		
	time	nodes	objects		time	nodes	objects		time	nodes	objects
a p 12 (1)	28	1	7198	a p 12 (2)	28	1	7227	a p 12 (3)	28	1	7247
a p 12 (4)	28	1	7412	a p 12 (5)	29	1	7191	a p 12 (6)	29	1	7234
a p 12 (7)	29	1	7240	a p 12 (8)	29	1	7246	a p 12 (9)	29	1	7260
a p 12 (10)	29	1	7282	a p 12 (11)	29	1	7282	a p 12 (12)	29	1	7290
a p 12 (13)	29	1	7294	a p 12 (14)	29	1	7294	a p 12 (15)	29	1	7299
a p 12 (16)	29	1	7303	a p 12 (17)	29	1	7314	a p 12 (18)	29	1	7316
a p 12 (19)	29	1	7322	a p 12 (20)	29	1	7328	a p 12 (21)	29	1	7345
a p 12 (22)	29	1	7346	a p 12 (23)	30	1	7226	a p 12 (24)	30	1	7251
a p 12 (25)	30	1	7267	a p 12 (26)	30	1	7269	a p 12 (27)	30	1	7275
a p 12 (28)	30	1	7287	a p 12 (29)	30	1	7289	a p 12 (30)	30	1	7294
a p 12 (31)	30	1	7295	a p 12 (32)	30	1	7301	a p 12 (33)	30	1	7312
a p 12 (34)	30	1	7313	a p 12 (35)	30	1	7319	a p 12 (36)	30	1	7322
a p 12 (37)	30	1	7323	a p 12 (38)	30	1	7327	a p 12 (39)	30	1	7329
a p 12 (40)	30	1	7330	a p 12 (41)	30	1	7339	a p 12 (42)	30	1	7342
a p 12 (43)	30	1	7343	a p 12 (44)	30	1	7349	a p 12 (45)	31	1	7252
a p 12 (46)	31	1	7278	a p 12 (47)	31	1	7291	a p 12 (48)	31	1	7292
a p 12 (49)	31	1	7303	a p 12 (50)	31	1	7303	a p 12 (51)	31	1	7304
a p 12 (52)	31	1	7306	a p 12 (53)	31	1	7310	a p 12 (54)	31	1	7314
a p 12 (55)	31	1	7318	a p 12 (56)	31	1	7321	a p 12 (57)	31	1	7324
a p 12 (58)	31	1	7324	a p 12 (59)	31	1	7329	a p 12 (60)	31	1	7337
a p 12 (61)	31	1	7339	a p 12 (62)	31	1	7339	a p 12 (63)	31	1	7348
a p 12 (64)	31	1	7355	a p 12 (65)	31	1	7367	a p 12 (66)	31	1	7391
a p 12 (67)	31	1	7407	a p 12 (68)	31	1	7453	a p 12 (69)	32	1	7283
a p 12 (70)	32	1	7309	a p 12 (71)	32	1	7373	a p 12 (72)	32	1	7376
a p 12 (73)	32	1	7420	a p 12 (74)	32	1	7450	a p 12 (75)	33	1	7269
a p 12 (76)	33	1	7486	a p 12 (77)	36	1	7279	a p 12 (78)	37	3	7994
a p 12 (79)	39	3	8925	a p 12 (80)	41	3	9065	a p 12 (81)	44	3	8270
a p 12 (82)	47	3	8533	a p 12 (83)	48	3	9387	a p 12 (84)	50	3	8345
a p 12 (85)	53	3	9600	a p 12 (86)	57	85	37768	a p 12 (87)	>120	-	-
a p 12 (88)	>120	-	-	a p 12 (89)	>120	-	-	a p 12 (90)	>120	-	-
a p 12 (91)	>120	-	-	a p 12 (92)	>120	-	-	a p 12 (93)	>120	-	-
a p 12 (94)	>120	-	-	a p 12 (95)	>120	-	-	a p 12 (96)	>120	-	-
a p 12 (97)	>120	-	-	a p 12 (98)	>120	-	-	a p 12 (99)	>120	-	-
a p 12 (100)	>120	-	-								

Table 5.7: Equivalence checking of add-stepper multiplier against simplified Wallace multiplier, with small random "tweak" to the former one

benchmark	basolver			benchmark	basolver			benchmark	basolver		
	time	nodes	objects		time	nodes	objects		time	nodes	objects
d p 12 (1)	28	1	7429	d p 12 (2)	28	1	7435	d p 12 (3)	28	1	7501
d p 12 (4)	28	1	7683	d p 12 (5)	29	1	7318	d p 12 (6)	29	1	7462
d p 12 (7)	29	1	7466	d p 12 (8)	29	1	7470	d p 12 (9)	29	1	7475
d p 12 (10)	29	1	7489	d p 12 (11)	29	1	7498	d p 12 (12)	29	1	7498
d p 12 (13)	29	1	7511	d p 12 (14)	29	1	7515	d p 12 (15)	29	1	7518
d p 12 (16)	29	1	7528	d p 12 (17)	29	1	7569	d p 12 (18)	29	1	7610
d p 12 (19)	30	1	7424	d p 12 (20)	30	1	7449	d p 12 (21)	30	1	7456
d p 12 (22)	30	1	7459	d p 12 (23)	30	1	7479	d p 12 (24)	30	1	7481
d p 12 (25)	30	1	7482	d p 12 (26)	30	1	7484	d p 12 (27)	30	1	7491
d p 12 (28)	30	1	7494	d p 12 (29)	30	1	7537	d p 12 (30)	30	1	7544
d p 12 (31)	30	1	7549	d p 12 (32)	30	1	7550	d p 12 (33)	30	1	7563
d p 12 (34)	30	1	7576	d p 12 (35)	30	1	7576	d p 12 (36)	30	1	7582
d p 12 (37)	30	1	7587	d p 12 (38)	30	1	7589	d p 12 (39)	30	1	7598
d p 12 (40)	30	1	7650	d p 12 (41)	30	1	7650	d p 12 (42)	30	1	7650
d p 12 (43)	30	1	7653	d p 12 (44)	30	1	7680	d p 12 (45)	30	1	7702
d p 12 (46)	30	1	7732	d p 12 (47)	31	1	7496	d p 12 (48)	31	1	7506
d p 12 (49)	31	1	7520	d p 12 (50)	31	1	7532	d p 12 (51)	31	1	7533
d p 12 (52)	31	1	7551	d p 12 (53)	31	1	7567	d p 12 (54)	31	1	7570
d p 12 (55)	31	1	7579	d p 12 (56)	31	1	7583	d p 12 (57)	31	1	7584
d p 12 (58)	31	1	7589	d p 12 (59)	31	1	7591	d p 12 (60)	31	1	7617
d p 12 (61)	31	1	7632	d p 12 (62)	31	1	7635	d p 12 (63)	31	1	7648
d p 12 (64)	31	1	7649	d p 12 (65)	31	1	7659	d p 12 (66)	31	1	7694
d p 12 (67)	31	1	7700	d p 12 (68)	31	1	7702	d p 12 (69)	31	1	7766
d p 12 (70)	32	1	7635	d p 12 (71)	32	1	7635	d p 12 (72)	32	1	7641
d p 12 (73)	32	1	7641	d p 12 (74)	32	1	7727	d p 12 (75)	32	1	7797
d p 12 (76)	33	1	7694	d p 12 (77)	33	1	7696	d p 12 (78)	35	3	8082
d p 12 (79)	35	3	8126	d p 12 (80)	37	3	8128	d p 12 (81)	40	3	8446
d p 12 (82)	48	3	10106	d p 12 (83)	52	3	8798	d p 12 (84)	52	3	9042
d p 12 (85)	55	3	10217	d p 12 (86)	>120	-	-	d p 12 (87)	>120	-	-
d p 12 (88)	>120	-	-	d p 12 (89)	>120	-	-	d p 12 (90)	>120	-	-
d p 12 (91)	>120	-	-	d p 12 (92)	>120	-	-	d p 12 (93)	>120	-	-
d p 12 (94)	>120	-	-	d p 12 (95)	>120	-	-	d p 12 (96)	>120	-	-
d p 12 (97)	>120	-	-	d p 12 (98)	>120	-	-	d p 12 (99)	>120	-	-
d p 12 (100)	>120	-	-								

Table 5.8: Equivalence checking of diagonal multiplier against simplified Wallace multiplier, with small random "tweak" to the former one

benchmark	basolver		satelite		minisat		zchaff		berkmin	
	time	objects	time	nodes	time	nodes	time	nodes	time	nodes
B2a a 3	0	510	0	22	0	21	0	35	0	70
B2a a 4	0	1219	0	168	0	103	0	498	0	15
B2a a 5	1	2078	0	450	0	2262	0	1559	0	1557
B2a a 6	2	3176	0	6803	0	8170	2	8360	0	7140
B2a a 7	3	4584	2	50881	1	15869	32	39576	10	31946
B2a a 8	5	6155	1	24240	7	53819	292	200204	90	101899
B2a a 9	8	8066	0	7070	8	44146	473	340988	807	379677
B2a a 10	11	10240	79	445743	109	309858	>3600	-	>3600	-
B2a a 11	16	12785	151	673136	2903	4065023	>3600	-	>3600	-
B2a a 12	22	15086	0	3080	3600	3900253	>3600	-	>3600	-
B2a a 13	28	18570	>3600	-	>3600	-	>3600	-	>3600	-
B2a a 14	39	21288	DNS	-	DNS	-	DNS	-	DNS	-
B2a a 15	49	24681	DNS	-	DNS	-	DNS	-	DNS	-
B2a a 16	65	29149	DNS	-	DNS	-	DNS	-	DNS	-
B2a a 32	1213	123184	DNS	-	DNS	-	DNS	-	DNS	-
B2a a 40	3040	197397	DNS	-	DNS	-	DNS	-	DNS	-
B2a d 3	0	1179	0	101	0	89	0	33	0	93
B2a d 4	1	2722	0	626	0	557	0	271	0	187
B2a d 5	2	4765	0	1240	0	2649	0	1919	0	297
B2a d 6	4	7359	0	7780	0	9086	0	4230	0	7206
B2a d 7	9	11629	0	14738	2	26332	3	8521	9	30631
B2a d 8	14	15481	17	228454	5	32545	>3600	-	145	139706
B2a d 9	23	21733	11	113868	23	91773	>3600	-	310	222249
B2a d 10	34	26545	22	153873	201	490029	0	2247	468	307970
B2a d 11	49	34246	2616	6738055	1520	2421106	>3600	-	>3600	-
B2a d 12	72	42155	442	1131896	>3600	-	>3600	-	3255	1170516
B2a d 13	96	50260	>3600	-	>3600	-	>3600	-	>3600	-
B2a d 14	136	63230	DNS	-	DNS	-	DNS	-	DNS	-
B2a d 15	178	70628	DNS	-	DNS	-	DNS	-	DNS	-
B2a d 16	236	88771	DNS	-	DNS	-	DNS	-	DNS	-
B2a d 28	2382	360712	DNS	-	DNS	-	DNS	-	DNS	-
B2a ma 3	0	1709	0	31	0	47	0	72	0	39
B2a ma 4	1	3810	0	52	0	281	0	330	0	12
B2a ma 5	3	7353	0	434	0	619	0	1849	0	69
B2a ma 6	6	11719	0	10782	0	2036	0	1541	1	8161
B2a ma 7	11	18090	0	11657	0	2367	44	50786	10	30221
B2a ma 8	17	26016	1	15872	1	11170	797	357416	131	138169
B2a ma 9	25	34269	5	60172	1	8857	3177	956444	1113	496356
B2a ma 10	38	45219	18	141835	148	289116	>3600	-	1203	561467
B2a ma 11	53	59270	239	959855	1260	1628803	>3600	-	3288	1224246
B2a ma 12	73	73510	1379	3764141	3256	2888311	>3600	-	>3600	-
B2a ma 13	102	90006	>3600	-	>3600	-	>3600	-	>3600	-
B2a ma 14	126	104419	DNS	-	DNS	-	DNS	-	DNS	-
B2a ma 15	169	126155	DNS	-	DNS	-	DNS	-	DNS	-
B2a ma 16	232	158446	DNS	-	DNS	-	DNS	-	DNS	-
B2a ma 24	1217	478186	DNS	-	DNS	-	DNS	-	DNS	-
B2a ma 32	3372	1000347	DNS	-	DNS	-	DNS	-	DNS	-
B2ma a 3	0	1459	0	101	0	58	0	80	0	42
B2ma a 4	1	3835	0	68	0	88	0	309	0	310
B2ma a 5	3	7048	0	332	0	1127	0	2069	0	1367
B2ma a 6	6	12015	0	7516	0	4243	0	2516	0	7152
B2ma a 7	9	16585	0	988	0	1017	34	46873	11	30259
B2ma a 8	16	25408	0	6215	0	6988	453	257358	134	126218
B2ma a 9	23	36445	2	31987	3	18636	484	324268	637	357764
B2ma a 10	31	44446	16	126163	102	190489	>3600	-	67	93221
B2ma a 11	45	58850	913	3075786	210	310059	>3600	-	2786	1076871
B2ma a 12	65	73958	>3600	-	>3600	-	>3600	-	>3600	-
B2ma a 13	84	89440	>3600	-	>3600	-	>3600	-	>3600	-
B2ma a 14	114	110610	DNS	-	DNS	-	DNS	-	DNS	-
B2ma a 15	146	132484	DNS	-	DNS	-	DNS	-	DNS	-
B2ma a 16	183	152995	DNS	-	DNS	-	DNS	-	DNS	-
B2ma a 26	1313	595236	DNS	-	DNS	-	DNS	-	DNS	-
B2ma a 32	3030	1008537	DNS	-	DNS	-	DNS	-	DNS	-

Table 5.9: Equivalence checking of buggy multipliers with one faulty pair of inputs

Chapter 6

Conclusion

We implemented a propositional solver based on a mixed boolean-algebraic strategy that combines DPLL with algebraic inference rules.

We demonstrated that this strategy can produce automated proofs of equivalences between different multipliers. However, a nontrivial effort of engineers is required for introducing a multiplier based on a new idea to the framework: an invariant corresponding to a handmade proof must be formulated, and inference rules for inductive maintaining of this invariant must be designed. Then, hopefully a computer is able to compute the invariant for multiplier circuits of any reasonable number of bits and thus prove that two multipliers are equivalent. We believe that the same strategy is applicable not only to equivalence checking, but also to the verification of other properties of a circuit.

The solver has demonstrated a reasonably rigid behavior: when one of the circuits is slightly changed, frequently, `basolver` is still able to solve the equivalence checking problem formulated as a CNF (despite the invariant may be no longer satisfied). This works perfectly for almost identical multipliers (this may be not a big deal if one looks at the circuits themselves, but contemporary CNF solvers are helplessly stuck on such instances), and also works for many pairs of circuits similar to those for which we are able to prove the equivalence without changes. The strategy here is to split by variables that resist to the straightforward derivation. This behavior can be further improved if a more involved heuristic for choosing literals for splitting is used.

The main problems with our "inductive" strategy emerge when the invariant has a high degree. For example, a four-degree invariant for the equivalence between simplified Booth and add step multipliers did not enable us to produce an automated proof of this equivalence. There are two possible directions for fighting this problem: a more extensive use of disjunctions of equalities and switching from polynomial equalities to polynomial inequalities. It is, however, unclear whether natural complications concerning strengthening the proof system will permit a more efficient proof search. Another possible direction is an intelligent splitting of a problem into subproblems which may lower the proof degree.

Our experiments on buggy circuits demonstrated that while our solver is able to find severe bugs that make a circuit unusable (like an inverted gate, for example), it is much less efficient in it than general-purpose SAT solvers. However, if a bug appears rarely (and thus the equivalence checking formula has few satisfying assignments), the problem becomes easier for `basolver`.

The general advice following from our research is that research strategy can be very different depending on what multipliers are being checked and how far are they one from the other. It may be reasonable to implement a separate solver for each verification problem (for example, for a series of multiplier circuits based on the same idea). The framework we have created is suitable for building various combinations of rules and may serve as the basis for further elaborations on different circuit series. One such elaboration — for the Booth multipliers — we have implemented ourselves.

Bibliography

[Booth, 1951] A. D. Booth, A signed binary multiplication technique, *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236–240, 1951.

[Bryant, Chen, 2001] R. E. Bryant, Y.-A. Chen. Verification of arithmetic circuits using binary moment diagrams, *International Journal on Software Tools for Technology Transfer* 3(2), pp. 137–155, 2001.

[Davis et al., 1962] M. Davis, G. Logemann, and D. Loveland, A machine program for theorem-proving, *Communications of the ACM*, 5(7), pp. 394–397, 1962.

[Davis and Putnam, 1960] M. Davis and H. Putnam, A computing procedure for quantification theory, *J. of the ACM*, 7(3), pp. 201–215, 1960.

[del Val, 2000] Alvaro del Val, On 2-SAT and Renamable Horn, *Proc. 17th National Conference on Artificial Intelligence*, 2000.

[Wallace, 1964] C. S. Wallace, A Suggestion for a Fast Multiplexer, *IEEE Trans. Electron Comput.*, p. 14-17, Feb., 1964.

Appendix A: Circuit descriptions

A.1 Add-stepper multiplier

In this appendix we begin to describe formally the multipliers that we have provided equivalence proofs for. The ultimate goal of each subsection is to provide a CNF formula that describes a certain multiplier.

We begin with two simplest multiplication schemes that are depicted on Fig. A.1

Each cell in the schemes stands for a standard multiplexer, as depicted on Fig. A.2.

In the following formula we encode the simplest add-stepper multiplier that multiplies two numbers: $X = \{x_0, \ldots, x_N\}$ having $N+1$ bits and $Y = \{y_0, \ldots, y_M\}$ consisting of $M+1$ bits. To the right of each block of clauses we write equalities that correspond to this block of clauses (this notation will be common practice throughout this document).

1. The zeroth column is encoded by the following set of clauses:

$$\left.\begin{aligned} x_0 \vee \neg r_{00} \\ y_0 \vee \neg r_{00} \\ r_{00} \vee \neg x_0 \vee \neg y_0 \end{aligned}\right\} \quad r_{00} = x_0 \wedge y_0$$

$$\vdots$$

$$\left.\begin{aligned} x_N \vee \neg r_{N0} \\ y_0 \vee \neg r_{N0} \\ r_{N0} \vee \neg x_N \vee \neg y_0 \end{aligned}\right\} \quad r_{N0} = x_N \wedge y_0$$

2. The lowest cell in each of the following columns ($j = 1..M$) is given as follows:

$$\left.\begin{aligned} x_0 \vee \neg c_{0j} \\ y_j \vee \neg c_{0j} \\ c_{0j} \vee \neg x_0 \vee \neg y_j \end{aligned}\right\} \quad c_{0j} = x_0 \wedge y_j$$

$$\left.\begin{aligned} \neg c_{0j} \vee r_{1j-1} \vee r_{0j} \\ c_{0j} \vee \neg r_{1j-1} \vee r_{0j} \\ c_{0j} \vee r_{1j-1} \vee \neg r_{0j} \\ \neg c_{0j} \vee \neg r_{1j-1} \vee \neg r_{0j} \end{aligned}\right\} \quad r_{0j} = c_{0j} \oplus r_{1j-1}$$

$$\left.\begin{aligned} c_{0j} \vee \neg t_{0j} \\ r_{1j-1} \vee \neg t_{0j} \\ t_{0j} \vee \neg c_{0j} \vee \neg r_{1j-1} \end{aligned}\right\} \quad t_{0j} = c_{0j} \wedge r_{1j-1}$$

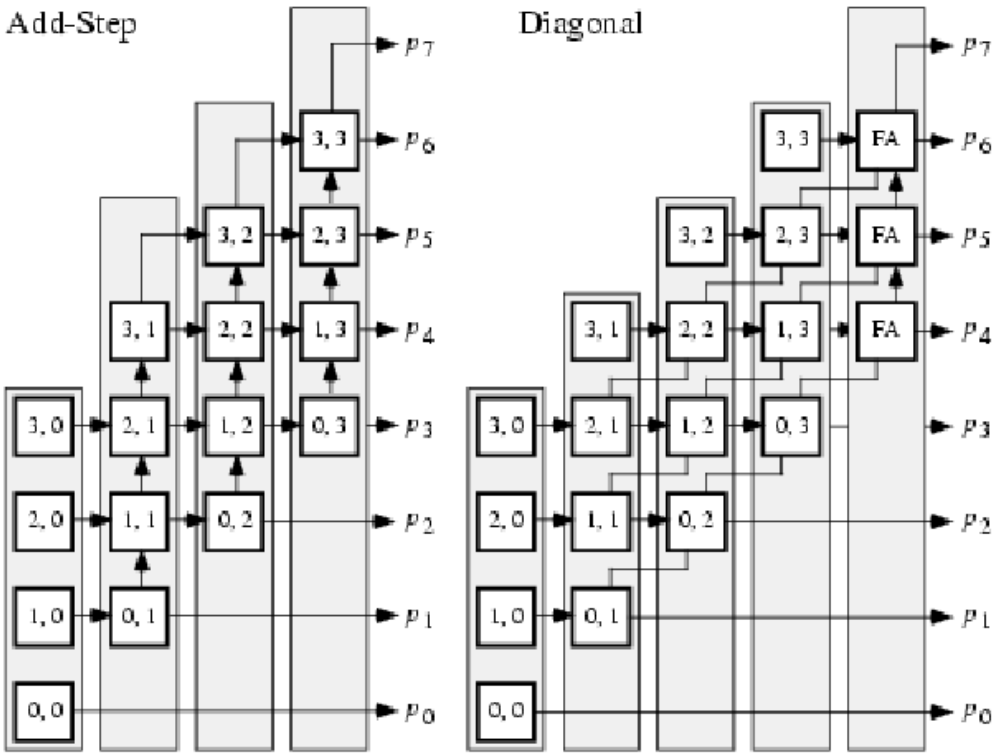

Figure A.1: Two standard 4-bit multipliers. The picture is taken from [Bryant, Chen, 2001]. We corrected a small typo in the article.

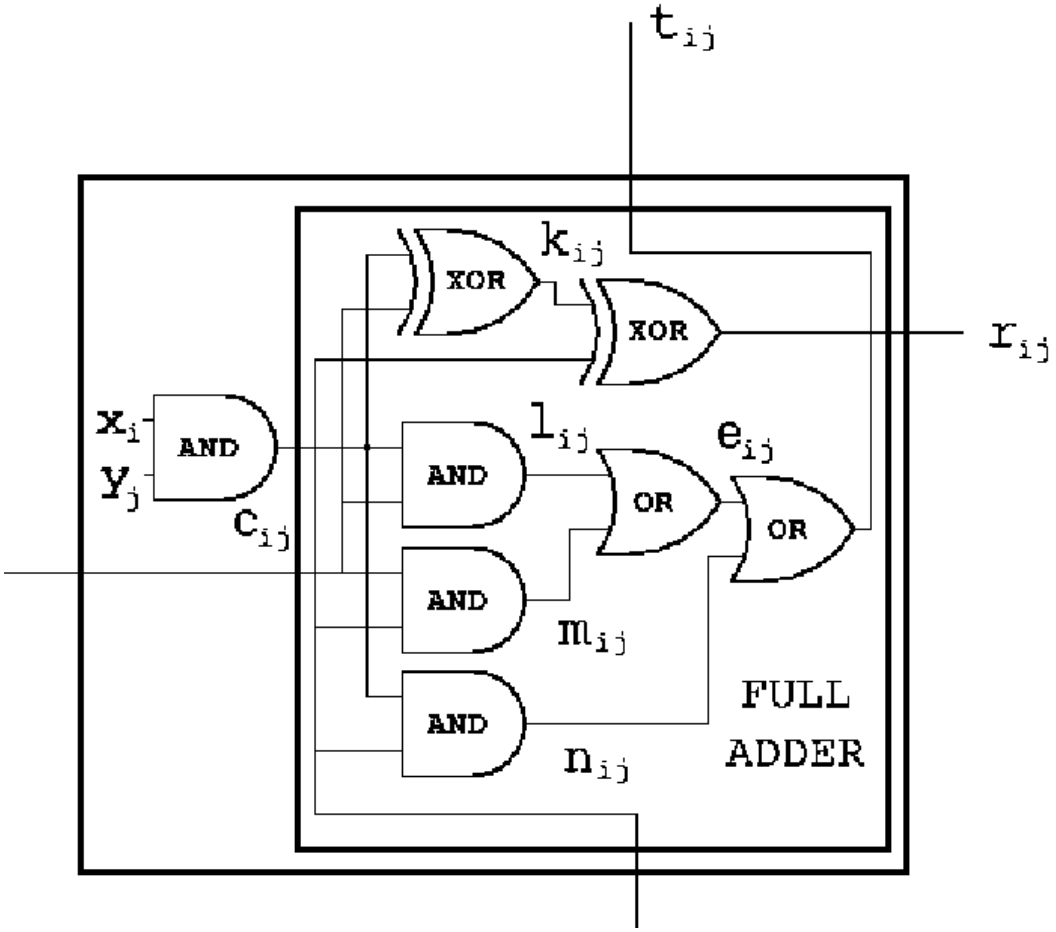

Figure A.2: Cell consisting of an AND gate and a full adder.

3. The upper cell of the first column is given by the following set of clauses:

$$\left.\begin{aligned} x_N \vee \neg c_{N1} \\ y_1 \vee \neg c_{N1} \\ c_{N1} \vee \neg x_N \vee \neg y_1 \end{aligned}\right\} \quad c_{N1} = x_N \wedge y_1$$

$$\left.\begin{aligned} \neg c_{N1} \vee t_{N-1,1} \vee r_{N1} \\ c_{N1} \vee \neg t_{N-1,1} \vee r_{N1} \\ c_{N1} \vee t_{N-1,1} \vee \neg r_{N1} \\ \neg c_{N1} \vee \neg t_{N-1,1} \vee \neg r_{N1} \end{aligned}\right\} \quad r_{N1} = c_{N1} \oplus t_{N-1,1}$$

$$\left.\begin{aligned} c_{N1} \vee \neg t_{N1} \\ t_{N-1,1} \vee \neg t_{N1} \\ t_{N1} \vee \neg c_{N1} \vee \neg t_{N-1,1} \end{aligned}\right\} \quad t_{N1} = c_{N1} \wedge t_{N-1,1}$$

4. The upper cells in the remaining columns ($j = 1..M$) may be encoded by the following sets of clauses:

$$\left.\begin{aligned} x_N \vee \neg c_{Nj} \\ y_j \vee \neg c_{Nj} \\ c_{Nj} \vee \neg x_N \vee \neg y_j \end{aligned}\right\} \quad c_{Nj} = x_N \wedge y_j$$

$$\left.\begin{aligned}\neg c_{Nj} \vee t_{N,j-1} \vee k_{Nj}\\ c_{Nj} \vee \neg t_{N,j-1} \vee k_{Nj}\\ c_{Nj} \vee t_{N,j-1} \vee \neg k_{Nj}\\ \neg c_{Nj} \vee \neg t_{N,j-1} \vee \neg k_{Nj}\end{aligned}\right\} \quad k_{Nj} = c_{Nj} \oplus t_{N,j-1}$$

$$\left.\begin{aligned}\neg k_{Nj} \vee t_{N-1,j} \vee r_{Nj}\\ k_{Nj} \vee \neg t_{N-1,j} \vee r_{Nj}\\ k_{Nj} \vee t_{N-1,j} \vee \neg r_{Nj}\\ \neg k_{Nj} \vee \neg t_{N-1,j} \vee \neg r_{Nj}\end{aligned}\right\} \quad r_{Nj} = k_{Nj} \oplus t_{N-1,j}$$

$$\left.\begin{aligned}c_{Nj} \vee \neg l_{Nj}\\ t_{N,j-1} \vee \neg l_{Nj}\\ l_{Nj} \vee \neg c_{Nj} \vee \neg t_{N,j-1}\end{aligned}\right\} \quad l_{Nj} = c_{Nj} \wedge t_{N,j-1}$$

$$\left.\begin{aligned}c_{Nj} \vee \neg m_{Nj}\\ t_{N-1,j} \vee \neg m_{Nj}\\ m_{Nj} \vee \neg c_{Nj} \vee \neg t_{N-1,j}\end{aligned}\right\} \quad m_{Nj} = c_{Nj} \wedge t_{N-1,j}$$

$$\left.\begin{aligned}t_{N,j-1} \vee \neg n_{Nj}\\ t_{N-1,j} \vee \neg n_{Nj}\\ n_{Nj} \vee \neg t_{N,j-1} \vee \neg t_{N-1,j}\end{aligned}\right\} \quad n_{Nj} = t_{N,j-1} \wedge t_{N-1,j}$$

$$\left.\begin{aligned}\neg l_{Nj} \vee e_{Nj}\\ \neg m_{Nj} \vee e_{Nj}\\ \neg e_{Nj} \vee l_{Nj} \vee m_{Nj}\end{aligned}\right\} \quad e_{Nj} = l_{Nj} \vee m_{Nj}$$

$$\left.\begin{aligned}\neg n_{Nj} \vee t_{Nj}\\ \neg e_{Nj} \vee t_{Nj}\\ \neg t_{Nj} \vee n_{Nj} \vee e_{Nj}\end{aligned}\right\} \quad t_{Nj} = n_{Nj} \vee e_{Nj}$$

5. Finally, each of the remaining cells (corresponding to the pairs $\{i,j\}_{i=1..N-1,j=1..M}$) is given by the following set of clauses:

$$\left.\begin{aligned}x_i \vee \neg c_{ij}\\ y_j \vee \neg c_{ij}\\ c_{ij} \vee \neg x_i \vee \neg y_j\end{aligned}\right\} \quad c_{ij} = x_i \wedge y_j$$

$$\left.\begin{aligned}\neg c_{ij} \vee r_{i+1,j-1} \vee k_{ij}\\ c_{ij} \vee \neg r_{i+1,j-1} \vee k_{ij}\\ c_{ij} \vee r_{i+1,j-1} \vee \neg k_{ij}\\ \neg c_{ij} \vee \neg r_{i+1,j-1} \vee \neg k_{ij}\end{aligned}\right\} \quad k_{ij} = c_{ij} \oplus r_{i+1,j-1}$$

$$\left.\begin{aligned}\neg k_{ij} \vee t_{i-1,j} \vee r_{ij}\\ k_{ij} \vee \neg t_{i-1,j} \vee r_{ij}\\ k_{ij} \vee t_{i-1,j} \vee \neg r_{ij}\\ \neg k_{ij} \vee \neg t_{i-1,j} \vee \neg r_{ij}\end{aligned}\right\} \quad r_{ij} = k_{ij} \oplus t_{i-1,j}$$

$$\left.\begin{aligned}c_{ij} \vee \neg l_{ij}\\ r_{i+1,j-1} \vee \neg l_{ij}\\ l_{ij} \vee \neg c_{ij} \vee \neg r_{i+1,j-1}\end{aligned}\right\} \quad l_{ij} = c_{ij} \wedge r_{i+1,j-1}$$

$$\left.\begin{aligned} c_{ij} \vee \neg m_{ij} \\ t_{i-1,j} \vee \neg m_{ij} \\ m_{ij} \vee \neg c_{ij} \vee \neg t_{i-1,j} \end{aligned}\right\} \quad m_{ij} = c_{ij} \wedge t_{i-1,j}$$

$$\left.\begin{aligned} r_{i+1,j-1} \vee \neg n_{ij} \\ t_{i-1,j} \vee \neg n_{ij} \\ n_{ij} \vee \neg r_{i+1,j-1} \vee \neg t_{i-1,j} \end{aligned}\right\} \quad n_{ij} = r_{i+1,j-1} \wedge t_{i-1,j}$$

$$\left.\begin{aligned} \neg l_{ij} \vee e_{ij} \\ \neg m_{ij} \vee e_{ij} \\ \neg e_{ij} \vee l_{ij} \vee m_{ij} \end{aligned}\right\} \quad e_{ij} = l_{ij} \vee m_{ij}$$

$$\left.\begin{aligned} \neg n_{ij} \vee t_{ij} \\ \neg e_{ij} \vee t_{ij} \\ \neg t_{ij} \vee n_{ij} \vee e_{ij} \end{aligned}\right\} \quad t_{ij} = n_{ij} \vee e_{ij}$$

6. In the formula given above certain variables correspond to certain outputs of the circuit:
   - $r_{0j}, 0 \leq j \leq M$ correspond to the first $M+1$ outputs of the multiplier — $p_i, 0 \leq i \leq M$;
   - $r_{iM}, 1 \leq i \leq N$ correspond to the next N outputs of the multiplier — $p_i, M+1 \leq i \leq M+N$;
   - p_{M+N+1} corresponds to t_{NM}.
7. We consider the case when one of the inputs consists of one bit as a special case. It should be treated separately.

A.2 Diagonal multiplier

As one can see on Fig. A.1, the idea of a diagonal multiplier is greatly similar to the add-stepper multiplier; the difference is in the way carry bits are propagated — they are propagated to the next level immediately, which allows to go through each column in one step, eliminating the need in calculating lower bits of a column in order to calculate upper bits.

A diagonal multiplier multiplying numbers $X = \{x_0, \ldots, x_N\}$ ($N+1$ bits) and $Y = \{y_0, \ldots, y_M\}$ ($M+1$ bits) may be encoded by the following formula:

1. The zeroth column is given by the following set of clauses:

$$\left.\begin{aligned} x_0 \vee \neg r_{00} \\ y_0 \vee \neg r_{00} \\ r_{00} \vee \neg x_0 \vee \neg y_0 \end{aligned}\right\} \quad r_{00} = x_0 \wedge y_0$$

$$\vdots$$

$$\left.\begin{aligned} x_N \vee \neg r_{N0} \\ y_0 \vee \neg r_{N0} \\ r_{N0} \vee \neg x_N \vee \neg y_0 \end{aligned}\right\} \quad r_{N0} = x_N \wedge y_0$$

2. The uppermost cell in each of the following columns ($j = 1..M$) is encoded as follows:

$$\left.\begin{aligned} x_N \vee \neg r_{Nj} \\ y_j \vee \neg r_{Nj} \\ r_{Nj} \vee \neg x_N \vee \neg y_j \end{aligned}\right\} \quad r_{Nj} = x_N \wedge y_j$$

3. The first column, except for the uppermost cell ($i = 0..N-1$), is given by the following clauses:

$$\left.\begin{aligned} x_i \vee \neg c_{i1} \\ y_1 \vee \neg c_{i1} \\ c_{i1} \vee \neg x_i \vee \neg y_1 \end{aligned}\right\} \quad c_{i1} = x_i \wedge y_1$$

$$\left.\begin{aligned} \neg c_{i1} \vee r_{i+1,0} \vee r_{i1} \\ c_{i1} \vee \neg r_{i+1,0} \vee r_{i1} \\ c_{i1} \vee r_{i+1,0} \vee \neg r_{i1} \\ \neg c_{i1} \vee \neg r_{i+1,0} \vee \neg r_{i1} \end{aligned}\right\} \quad r_{i1} = c_{i1} \oplus r_{i+1,0}$$

$$\left.\begin{aligned} c_{i1} \vee \neg t_{i1} \\ r_{i+1,0} \vee \neg t_{i1} \\ t_{i1} \vee \neg c_{i1} \vee \neg r_{i+1,0} \end{aligned}\right\} \quad t_{i1} = c_{i1} \wedge r_{i+1,0}$$

4. Finally, each of the rest of the cells (corresponding to the index pairs $\{i, j\}_{i=0..N-1, j=2..M}$) is encoded by the following set of clauses:

$$\left.\begin{aligned} x_i \vee \neg c_{ij} \\ y_j \vee \neg c_{ij} \\ c_{ij} \vee \neg x_i \vee \neg y_j \end{aligned}\right\} \quad c_{ij} = x_i \wedge y_j$$

$$\left.\begin{aligned} \neg c_{ij} \vee r_{i+1,j-1} \vee k_{ij} \\ c_{ij} \vee \neg r_{i+1,j-1} \vee k_{ij} \\ c_{ij} \vee r_{i+1,j-1} \vee \neg k_{ij} \\ \neg c_{ij} \vee \neg r_{i+1,j-1} \vee \neg k_{ij} \end{aligned}\right\} \quad k_{ij} = c_{ij} \oplus r_{i+1,j-1}$$

$$\left.\begin{aligned} \neg k_{ij} \vee t_{i,j-1} \vee r_{ij} \\ k_{ij} \vee \neg t_{i,j-1} \vee r_{ij} \\ k_{ij} \vee t_{i,j-1} \vee \neg r_{ij} \\ \neg k_{ij} \vee \neg t_{i,j-1} \vee \neg r_{ij} \end{aligned}\right\} \quad r_{ij} = k_{ij} \oplus t_{i,j-1}$$

$$\left.\begin{aligned} c_{ij} \vee \neg l_{ij} \\ r_{i+1,j-1} \vee \neg l_{ij} \\ l_{ij} \vee \neg c_{ij} \vee \neg r_{i+1,j-1} \end{aligned}\right\} \quad l_{ij} = c_{ij} \wedge r_{i+1,j-1}$$

$$\left.\begin{aligned} c_{ij} \vee \neg m_{ij} \\ t_{i,j-1} \vee \neg m_{ij} \\ m_{ij} \vee \neg c_{ij} \vee \neg t_{i,j-1} \end{aligned}\right\} \quad m_{ij} = c_{ij} \wedge t_{i,j-1}$$

$$\left.\begin{aligned} r_{i+1,j-1} \vee \neg n_{ij} \\ t_{i,j-1} \vee \neg n_{ij} \\ n_{ij} \vee \neg r_{i+1,j-1} \vee \neg t_{i,j-1} \end{aligned}\right\} \quad n_{ij} = r_{i+1,j-1} \wedge t_{i,j-1}$$

$$\left.\begin{aligned} \neg l_{ij} \vee e_{ij} \\ \neg m_{ij} \vee e_{ij} \\ \neg e_{ij} \vee l_{ij} \vee m_{ij} \end{aligned}\right\} \quad e_{ij} = l_{ij} \vee m_{ij}$$

$$\left.\begin{aligned} \neg n_{ij} \vee t_{ij} \\ \neg e_{ij} \vee t_{ij} \\ \neg t_{ij} \vee n_{ij} \vee e_{ij} \end{aligned}\right\} \quad t_{ij} = n_{ij} \vee e_{ij}$$

5. The last column (consisting of full adders — see Fig. A.1) is encoded by the following set of clauses ($1 \leq h \leq N-1$):

$$\left.\begin{aligned} \neg r_{1M} \vee t_{0M} \vee r_{0M+1} \\ r_{1M} \vee \neg t_{0M} \vee r_{0M+1} \\ r_{1M} \vee t_{0M} \vee \neg r_{0M+1} \\ \neg r_{1M} \vee \neg t_{0M} \vee \neg r_{0M+1} \end{aligned}\right\} \quad r_{0M+1} = r_{1M} \oplus t_{0M}$$

$$\left.\begin{aligned} r_{1M} \vee \neg t_{0M+1} \\ t_{0M} \vee \neg t_{0M+1} \\ t_{0M+1} \vee \neg r_{1M} \vee \neg t_{0M} \end{aligned}\right\} \quad t_{0M+1} = r_{1M} \wedge t_{0M}$$

$$\left.\begin{aligned} \neg r_{h+1,M} \vee t_{hM} \vee k_{hM+1} \\ r_{h+1,M} \vee \neg t_{hM} \vee k_{hM+1} \\ r_{h+1,M} \vee t_{hM} \vee \neg k_{hM+1} \\ \neg r_{h+1,M} \vee \neg t_{hM} \vee \neg k_{hM+1} \end{aligned}\right\} \quad k_{hM+1} = r_{h+1,M} \oplus t_{hM}$$

$$\left.\begin{aligned} \neg k_{hM+1} \vee t_{h-1,M+1} \vee r_{hM+1} \\ k_{hM+1} \vee \neg t_{h-1,M+1} \vee r_{hM+1} \\ k_{hM+1} \vee t_{h-1,M+1} \vee \neg r_{hM+1} \\ \neg k_{hM+1} \vee \neg t_{h-1,M+1} \vee \neg r_{hM+1} \end{aligned}\right\} \quad r_{hM+1} = k_{hM+1} \oplus t_{h-1,M+1}$$

$$\left.\begin{aligned} r_{h+1,M} \vee \neg l_{hM+1} \\ t_{hM} \vee \neg l_{hM+1} \\ l_{hM+1} \vee \neg r_{h+1,M} \vee \neg t_{hM} \end{aligned}\right\} \quad l_{hM+1} = r_{h+1,M} \wedge t_{hM}$$

$$\left.\begin{aligned} r_{h+1,M} \vee \neg m_{hM+1} \\ t_{h-1,M+1} \vee \neg m_{hM+1} \\ m_{hM+1} \vee \neg r_{h+1,M} \vee \neg t_{h-1,M+1} \end{aligned}\right\} \quad m_{hM+1} = r_{h+1,M} \wedge t_{h-1,M+1}$$

$$\left.\begin{aligned} t_{hM} \vee \neg n_{hM+1} \\ t_{h-1,M+1} \vee \neg n_{hM+1} \\ n_{hM+1} \vee \neg t_{hM} \vee \neg t_{h-1,M+1} \end{aligned}\right\} \quad n_{hM+1} = t_{hM} \wedge t_{h-1,M+1}$$

$$\left.\begin{aligned} \neg l_{hM+1} \vee e_{hM+1} \\ \neg m_{hM+1} \vee e_{hM+1} \\ \neg e_{hM+1} \vee l_{hM+1} \vee m_{hM+1} \end{aligned}\right\} \quad e_{hM+1} = l_{hM+1} \vee m_{hM+1}$$

$$\left.\begin{aligned} \neg n_{hM+1} \vee t_{hM+1} \\ \neg e_{hM+1} \vee t_{hM+1} \\ \neg t_{hM+1} \vee n_{hM+1} \vee e_{hM+1} \end{aligned}\right\} \quad t_{hM+1} = n_{hM+1} \vee e_{hM+1}$$

6. In the formula given above certain variables correspond to certain outputs of the circuit:
   - $r_{0j}, 0 \leq j \leq M$ correspond to the first $M+1$ outputs of the multiplier — $p_i, 0 \leq i \leq M$;
   - $r_{h,M+1}, 0 \leq h \leq N-1$ correspond to the next N outputs of the multiplier — $p_i, M+1 \leq i \leq M+N$;
   - p_{M+N+1} corresponds to $t_{N-1,M+1}$. $r_{0j}, 0 \leq j \leq M$ correspond to the first $M+1$
7. We consider the case when one of the inputs consists of one bit as a special case. It should be treated separately.

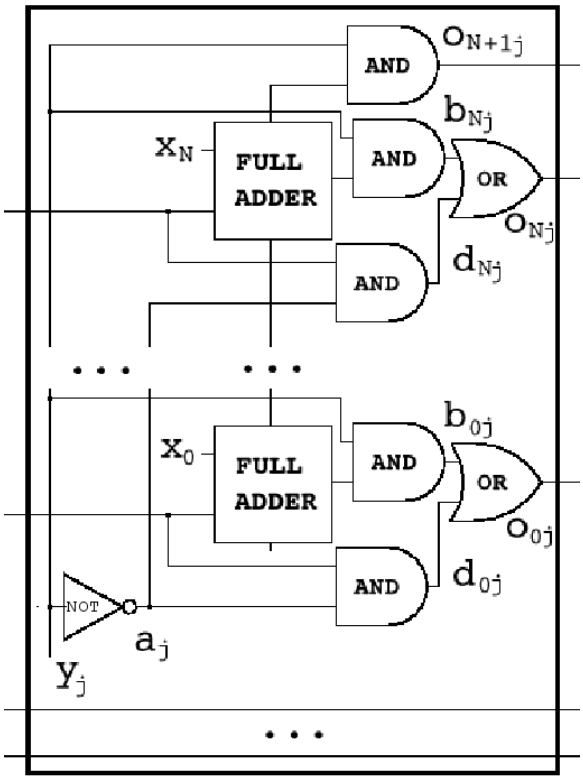

Figure A.3: Optimized column of the add-stepper multiplier.

A.3 Modified add-stepper multiplier

Modified add-stepper multiplier is obtained from add-stepper multiplier by optimizing its columns (see Fig. A.3). Namely, if the corresponding bit y_j is equal to 0, then all carry bits from the previous column are just transfered to the next level (without transferring through full adders). This idea is also used in Booth multiplier.

A diagonal multiplier multiplying numbers $X = \{x_0, \ldots, x_N\}$ ($N+1$ bits) and $Y = \{y_0, \ldots, y_M\}$ ($M+1$ bits) may be encoded by the following formula:

0. For each variable y_j, $0 \leq j \leq M$ we add the equality

$$\left.\begin{array}{r} y_j \vee a_j \\ \neg y_j \vee \neg a_j \end{array}\right\} \qquad a_j = \neg y_j$$

1. The zeroth column is given by the following set of clauses:

$$\left.\begin{array}{r} x_0 \vee \neg o_{00} \\ y_0 \vee \neg o_{00} \\ o_{00} \vee \neg x_0 \vee \neg y_0 \end{array}\right\} \qquad o_{00} = x_0 \wedge y_0$$

$$\vdots$$

$$\left.\begin{array}{r} x_N \vee \neg o_{N0} \\ y_0 \vee \neg o_{N0} \\ o_{N0} \vee \neg x_N \vee \neg y_0 \end{array}\right\} \qquad o_{N0} = x_N \wedge y_0$$

2. The lowest cell in all further ($j = 1..M$) columns is given by:

$$\left.\begin{aligned} \neg x_0 \vee o_{1j-1} \vee r_{0j} \\ x_0 \vee \neg o_{1j-1} \vee r_{0j} \\ x_0 \vee o_{1j-1} \vee \neg r_{0j} \\ \neg x_0 \vee \neg o_{1j-1} \vee \neg r_{0j} \end{aligned}\right\} \quad r_{0j} = x_0 \oplus o_{1j-1}$$

$$\left.\begin{aligned} x_0 \vee \neg t_{0j} \\ o_{1j-1} \vee \neg t_{0j} \\ t_{0j} \vee \neg x_0 \vee \neg o_{1j-1} \end{aligned}\right\} \quad t_{0j} = x_0 \wedge o_{1j-1}$$

$$\left.\begin{aligned} r_{0j} \vee \neg b_{0j} \\ y_j \vee \neg b_{0j} \\ b_{0j} \vee \neg r_{0j} \vee \neg y_j \end{aligned}\right\} \quad b_{0j} = r_{0j} \wedge y_j$$

$$\left.\begin{aligned} o_{1j-1} \vee \neg d_{0j} \\ a_j \vee \neg d_{0j} \\ d_{0j} \vee \neg o_{1j-1} \vee \neg a_j \end{aligned}\right\} \quad d_{0j} = o_{1j-1} \wedge a_j$$

$$\left.\begin{aligned} \neg d_{0j} \vee o_{0j} \\ \neg b_{0j} \vee o_{0j} \\ o_{0j} \vee d_{0j} \vee b_{0j} \end{aligned}\right\} \quad o_{0j} = d_{0j} \vee b_{0j}$$

3. The uppermost cell in the first column is encoded as:

$$\left.\begin{aligned} \neg x_N \vee t_{N-1,1} \vee r_{N1} \\ x_N \vee \neg t_{N-1,1} \vee r_{N1} \\ x_N \vee t_{N-1,1} \vee \neg r_{N1} \\ \neg x_N \vee \neg t_{N-1,1} \vee \neg r_{N1} \end{aligned}\right\} \quad r_{N1} = x_N \oplus t_{N-1,1}$$

$$\left.\begin{aligned} x_N \vee \neg t_{N1} \\ t_{N-1,1} \vee \neg t_{N1} \\ t_{N1} \vee \neg x_N \vee \neg t_{N-1,1} \end{aligned}\right\} \quad t_{N1} = x_N \wedge t_{N-1,1}$$

$$\left.\begin{aligned} r_{N1} \vee \neg o_{N1} \\ y_1 \vee \neg o_{N1} \\ o_{N1} \vee \neg r_{N1} \vee \neg y_1 \end{aligned}\right\} \quad o_{N1} = t_{N1} \wedge y_1$$

4. Let us encode all upper outputs in all columns except for the first column ($1 \leq j \leq M$):

$$\left.\begin{aligned} t_{Nj} \vee \neg o_{N+1j} \\ y_j \vee \neg o_{N+1j} \\ o_{N+1j} \vee \neg t_{Nj} \quad \vee \neg y_j \end{aligned}\right\} \quad o_{N+1j} = t_{Nj} \wedge y_j$$

5. All remaining cells (for the set of pairs $\{i,j\}_{i=1..N,j=1..M} \setminus \{1,N\}$) are given by the following set of clauses:

$$\left.\begin{aligned} \neg x_i \vee o_{i+1,j-1} \vee k_{ij} \\ x_i \vee \neg o_{i+1,j-1} \vee k_{ij} \\ x_i \vee o_{i+1,j-1} \vee \neg k_{ij} \\ \neg x_i \vee \neg o_{i+1,j-1} \vee \neg k_{ij} \end{aligned}\right\} \quad k_{ij} = x_i \oplus o_{i+1,j-1}$$

$$\left.\begin{array}{r}\neg k_{ij} \vee t_{i-1,j} \vee r_{ij} \\ k_{ij} \vee \neg t_{i-1,j} \vee r_{ij} \\ k_{ij} \vee t_{i-1,j} \vee \neg r_{ij} \\ \neg k_{ij} \vee \neg t_{i-1,j} \vee \neg r_{ij}\end{array}\right\} \quad r_{ij} = k_{ij} \oplus t_{i-1,j}$$

$$\left.\begin{array}{r}x_i \vee \neg l_{ij} \\ o_{i+1,j-1} \vee \neg l_{ij} \\ l_{ij} \vee \neg x_i \vee \neg o_{i+1,j-1}\end{array}\right\} \quad l_{ij} = x_i \wedge o_{i+1,j-1}$$

$$\left.\begin{array}{r}x_i \vee \neg m_{ij} \\ t_{i-1,j} \vee \neg m_{ij} \\ m_{ij} \vee \neg x_i \vee \neg t_{i-1,j}\end{array}\right\} \quad m_{ij} = x_i \wedge t_{i-1,j}$$

$$\left.\begin{array}{r}o_{i+1,j-1} \vee \neg n_{ij} \\ t_{i-1,j} \vee \neg n_{ij} \\ n_{ij} \vee \neg o_{i+1,j-1} \vee \neg t_{i-1,j}\end{array}\right\} \quad n_{ij} = o_{i+1,j-1} \wedge t_{i-1,j}$$

$$\left.\begin{array}{r}\neg l_{ij} \vee e_{ij} \\ \neg m_{ij} \vee e_{ij} \\ \neg e_{ij} \vee l_{ij} \vee m_{ij}\end{array}\right\} \quad e_{ij} = l_{ij} \vee m_{ij}$$

$$\left.\begin{array}{r}\neg n_{ij} \vee t_{ij} \\ \neg e_{ij} \vee t_{ij} \\ \neg t_{ij} \vee n_{ij} \vee e_{ij}\end{array}\right\} \quad t_{ij} = n_{ij} \vee e_{ij}$$

$$\left.\begin{array}{r}r_{ij} \vee \neg b_{ij} \\ y_j \vee \neg b_{ij} \\ b_{ij} \vee \neg r_{ij} \vee \neg y_j\end{array}\right\} \quad b_{ij} = r_{ij} \wedge y_j$$

$$\left.\begin{array}{r}o_{i+1,j-1} \vee \neg d_{ij} \\ a_j \vee \neg d_{ij} \\ d_{ij} \vee \neg o_{i+1,j-1} \vee \neg a_j\end{array}\right\} \quad d_{ij} = o_{i+1,j-1} \wedge a_j$$

$$\left.\begin{array}{r}\neg d_{ij} \vee o_{ij} \\ \neg b_{ij} \vee o_{ij} \\ o_{ij} \vee d_{ij} \vee b_{ij}\end{array}\right\} \quad o_{ij} = d_{ij} \vee b_{ij}$$

6. The first $M+1$ outputs of the multiplier $p_i, 0 \leq i \leq M$ correspond to $o_{0j}, 0 \leq j \leq M$, the next $N+1$ outputs of the multiplier $p_i, M+1 \leq i \leq M+N+1$ correspond to $o_{iM}, 1 \leq i \leq N+1$.

A.4 Simplified Wallace tree (parallel) multiplier

This kind of multiplier was first described in [Wallace, 1964]. The main idea of the suggested method is in parallel addition of partial products ($x_i \cdot Y$). The simplified[1] Wallace multiplier adds n $2n$-bit numbers in parallel. Fig. A.4 shows the structure of a Wallace tree adder. CSA in the picture indicates a carry-save adder having three multi-bit inputs and two multi-bit outputs.

Let us multiply X consisting of $n+1$ bits by Y consisting of $m+1$ bits. The multiplier's circuit is constructed by an inductive process, level by level, as follows:

[1] One could use parallel addition of the last two numbers to fully parallelize the multiplication; this is *not* the case for the multiplier that we study here.

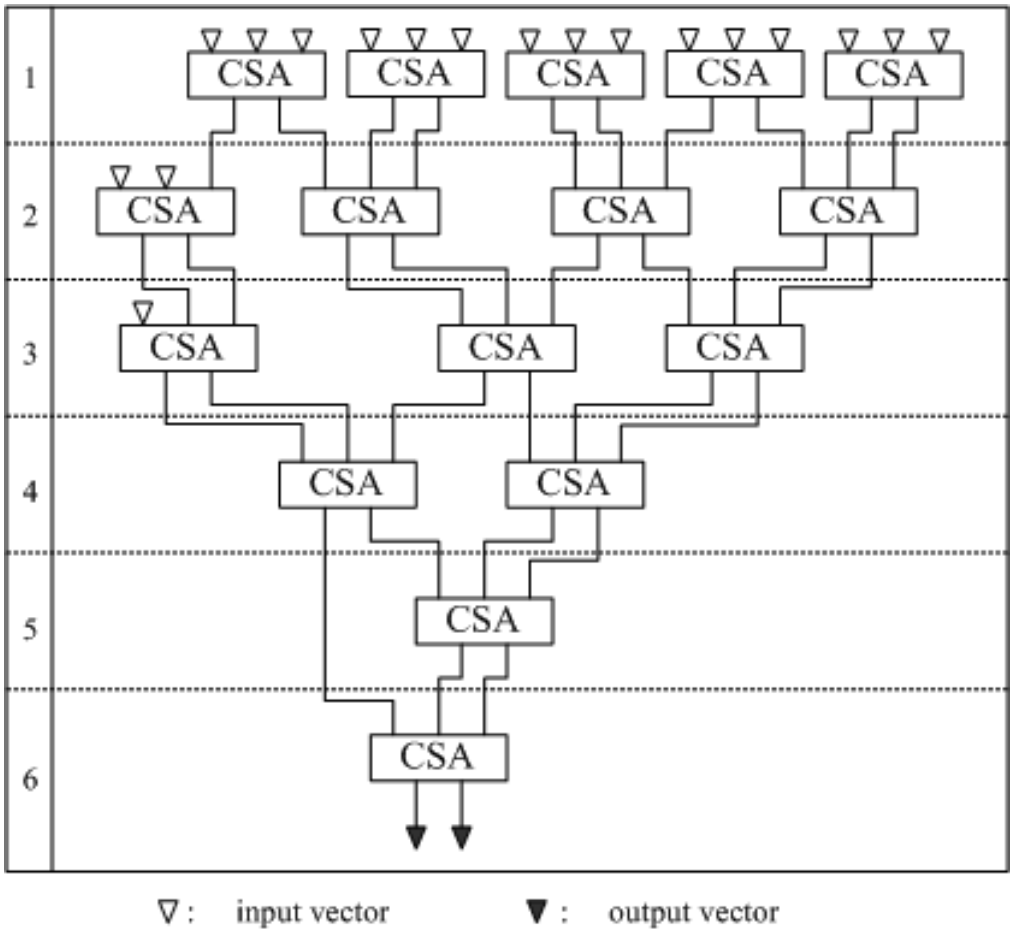

Figure A.4: Wallace tree adder. The picture is taken from `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html`

1. On the zeroth level we have the results of multiplying each of the bits x_i by Y:

$$c_{ij} = x_i y_j, \qquad \text{where } 0 \le i \le n,\ 0 \le j \le m.$$

   There are $n+1$ of the bitwise products C_i.

2. Suppose that the numbers we need to sum up on a given level are written down as a list S_n. Let us create a new list S_{n+1}, where for each triple of the numbers that were on places $3i$, $3i+1$, $3i+2$ in the S_n list we put into places $2i$, $2i+1$ their parallel sum (we shall describe this notion below). The remaining one or two numbers (that are left if the size of S_n is not divisible by 3) are simply pushed through to the back of S_{n+1}.

3. A parallel summator works as follows. Let the original triple be $A = \{\underbrace{0,\ldots,0}_{A_0}, a_{A_0+1},\ldots,a_{|A|}\}$, $B = \{\underbrace{0,\ldots,0}_{B_0}, b_{B_0+1},\ldots,b_{|B|}\}$, $C = \{\underbrace{0,\ldots,0}_{C_0}, c_{C_0+1},\ldots,c_{|C|}\}$. We shall denote the output (two numbers) by $R = \{\underbrace{0,\ldots,0}_{R_0}, r_{R_0+1},\ldots,r_{|R|}\}$ and $T = \{\underbrace{0,\ldots,0}_{T_0}, t_{T_0+1},\ldots,t_{|T|}\}$, where $R_0 = \min(A_0, B_0, C_0)$, $T_0 = \min(\{A_0, B_0, C_0\} \backslash \min(A_0, B_0, C_0)) + 1$ (mean of the A_0, B_0, C_0 increased by 1), $|R| = \max(|A|, |B|, |C|)$, and $|T| = \min(\{|A|, |B|, |C|\} \setminus \min(|A|, |B|, |C|)) + 1$ (mean of the $|A|, |B|, |C|$ increased by 1).

(a) Let $A_0 \le B_0 \le C_0$ (otherwise we may rename the numbers) and $\min(|A|, |B|, |C|) > C_0$ (otherwise we may make one number out of two). Then, for $A_0 \le j \le B_0$,

$$\left.\begin{aligned} r_j \vee \neg a_j \\ \neg r_j \vee a_j \end{aligned}\right\} \qquad r_j = a_j.$$

(b) For $B_0 \le j < C_0$ we should sum up two nonzero bits:

$$\left.\begin{aligned} \neg a_j \vee b_j \vee r_j \\ a_j \vee \neg b_j \vee r_j \\ a_j \vee b_j \vee \neg r_j \\ \neg a_j \vee \neg b_j \vee \neg r_j \end{aligned}\right\} \qquad r_j = a_j \oplus b_j$$

$$\left.\begin{aligned} a_j \vee \neg t_{j+1} \\ b_j \vee \neg t_{j+1} \\ t_{j+1} \vee \neg a_j \vee \neg b_j \end{aligned}\right\} \qquad t_{j+1} = a_j \wedge b_j.$$

(c) For $C_0 \le j < \min(|A|, |B|, |C|)$ we sum up three bits:

$$\left.\begin{aligned} \neg a_j \vee b_j \vee k_j \\ a_j \vee \neg b_j \vee k_j \\ a_j \vee b_j \vee \neg k_j \\ \neg a_j \vee \neg b_j \vee \neg k_j \end{aligned}\right\} \qquad k_j = a_j \oplus b_j$$

$$\left.\begin{aligned} \neg c_j \vee k_j \vee r_j \\ c_j \vee \neg k_j \vee r_j \\ c_j \vee k_j \vee \neg r_j \\ \neg c_j \vee \neg k_j \vee \neg r_j \end{aligned}\right\} \qquad r_j = c_j \oplus k_j$$

$$\left.\begin{aligned} a_j \vee \neg l_j \\ b_j \vee \neg l_j \\ l_j \vee \neg a_j \vee \neg b_j \end{aligned}\right\} \qquad l_j = a_j \wedge b_j$$

$$\left.\begin{aligned} a_j \vee \neg l_j \\ c_j \vee \neg l_j \\ l_j \vee \neg a_j \vee \neg c_j \end{aligned}\right\} \qquad m_j = a_j \wedge c_j$$

$$\left.\begin{aligned} c_j \vee \neg n_j \\ b_j \vee \neg n_j \\ n_j \vee \neg c_j \vee \neg b_j \end{aligned}\right\} \qquad n_j = c_j \wedge b_j$$

$$\left.\begin{aligned} \neg l_j \vee e_j \\ \neg m_j \vee e_j \\ \neg e_j \vee l_j \vee m_j \end{aligned}\right\} \qquad e_j = l_j \vee m_j$$

$$\left.\begin{aligned} \neg n_j \vee t_{j+1} \\ \neg e_j \vee t_{j+1} \\ \neg t_{j+1} \vee n_j \vee e_j \end{aligned}\right\} \qquad t_{j+1} = n_j \vee e_j.$$

(d) Let $|A| \le |B| \le |C|$ (otherwise we may rename the numbers again without loss of generality). Then, for $|A| \le j < |B|$,

$$\left.\begin{aligned} \neg c_j \vee b_j \vee r_j \\ c_j \vee \neg b_j \vee r_j \\ c_j \vee b_j \vee \neg r_j \\ \neg c_j \vee \neg b_j \vee \neg r_j \end{aligned}\right\} \quad r_j = c_j \oplus b_j$$

$$\left.\begin{aligned} c_j \vee \neg t_{j+1} \\ b_j \vee \neg t_{j+1} \\ t_{j+1} \vee \neg c_j \vee \neg b_j \end{aligned}\right\} \quad t_{j+1} = c_j \wedge b_j.$$

(e) For $|B| \le j < |C|$:

$$\left.\begin{aligned} r_j \vee \neg c_j \\ \neg r_j \vee c_j \end{aligned}\right\} \quad r_j = c_j.$$

4. If S_N consists of two numbers, we add them by a simple summator. Let the input numbers be $A = \{\underbrace{0, \ldots, 0}_{A_0}, a_{A_0+1}, \ldots, a_{|A|}\}$, $B = \{\underbrace{0, \ldots, 0}_{B_0}, b_{B_0+1}, \ldots, b_{|B|}\}$. We denote the output by $R = \{\underbrace{0, \ldots, 0}_{R_0}, r_{R_0+1}, \ldots, r_{|R|}\}$, where $R_0 = \min(A_0, B_0)$, $|R| = \max(|A|, |B|) + 1$ (let $A_0 \le B_0$, $|A| \le |B|$):

(a) For $A_0 \le j < B_0$:

$$\left.\begin{aligned} r_j \vee \neg a_j \\ \neg r_j \vee a_j \end{aligned}\right\} \quad r_j = a_j$$

(b) For $j = B_0$ (a somewhat special case):

$$\left.\begin{aligned} \neg a_j \vee b_j \vee r_j \\ a_j \vee \neg b_j \vee r_j \\ a_j \vee b_j \vee \neg r_j \\ \neg a_j \vee \neg b_j \vee \neg r_j \end{aligned}\right\} \quad r_j = a_j \oplus b_j$$

$$\left.\begin{aligned} a_j \vee \neg t_{j+1} \\ b_j \vee \neg t_{j+1} \\ t_{j+1} \vee \neg a_j \vee \neg b_j \end{aligned}\right\} \quad t_{j+1} = a_j \wedge b_j$$

(c) The usual summation with carry bits (for $B_0 < j < |A|$):

$$\left.\begin{aligned} \neg a_j \vee b_j \vee k_j \\ a_j \vee \neg b_j \vee k_j \\ a_j \vee b_j \vee \neg k_j \\ \neg a_j \vee \neg b_j \vee \neg k_j \end{aligned}\right\} \quad k_j = a_j \oplus b_j$$

$$\left.\begin{aligned} \neg t_j \vee k_j \vee r_j \\ t_j \vee \neg k_j \vee r_j \\ t_j \vee k_j \vee \neg r_j \\ \neg t_j \vee \neg k_j \vee \neg r_j \end{aligned}\right\} \quad r_j = t_j \oplus k_j$$

$$\left.\begin{aligned} a_j \vee \neg l_j \\ b_j \vee \neg l_j \\ l_j \vee \neg a_j \vee \neg b_j \end{aligned}\right\} \quad l_j = a_j \wedge b_j$$

$$\left.\begin{aligned} a_j \vee \neg l_j \\ t_j \vee \neg l_j \\ l_j \vee \neg a_j \vee \neg t_j \end{aligned}\right\} \quad m_j = a_j \wedge t_j$$

$$\left.\begin{aligned} t_j \vee \neg n_j \\ b_j \vee \neg n_j \\ n_j \vee \neg t_j \vee \neg b_j \end{aligned}\right\} \quad n_j = t_j \wedge b_j$$

$$\left.\begin{aligned} \neg l_j \vee e_j \\ \neg m_j \vee e_j \\ \neg e_j \vee l_j \vee m_j \end{aligned}\right\} \quad e_j = l_j \vee m_j$$

$$\left.\begin{aligned} \neg n_j \vee t_{j+1} \\ \neg e_j \vee t_{j+1} \\ \neg t_{j+1} \vee n_j \vee e_j \end{aligned}\right\} \quad t_{j+1} = n_j \vee e_j.$$

(d) For $|A| < j \leq |B|$:

$$\left.\begin{aligned} \neg t_j \vee b_j \vee r_j \\ t_j \vee \neg b_j \vee r_j \\ t_j \vee b_j \vee \neg r_j \\ \neg t_j \vee \neg b_j \vee \neg r_j \end{aligned}\right\} \quad r_j = t_j \oplus b_j$$

$$\left.\begin{aligned} t_j \vee \neg t_{j+1} \\ b_j \vee \neg t_{j+1} \\ t_{j+1} \vee \neg t_j \vee \neg b_j \end{aligned}\right\} \quad t_{j+1} = t_j \wedge b_j$$

(e) The last cell, $j = |B| + 1$:

$$\left.\begin{aligned} r_j \vee \neg t_j \\ \neg r_j \vee t_j \end{aligned}\right\} \quad r_j = t_j.$$

A.5 Booth and simplified Booth multipliers

The idea of the multiplication algorithm presented here dates back to [Booth, 1951]. We shall first present the basic idea, and then describe two different implementations, one being a simplified version of another.

One of the basic ideas that lie behind Booth multipliers is a shortcut for the basic add-stepper circuit. It is obvious that, if a certain bit of one of the factors is zero, multiplication by that bit may be greatly simplified. As we have seen above, modified add-stepper circuits are based on this simple remark.

Booth's another idea was to use a well-known trick (usually named after Abel) which allows to split a sum into sum of differences of neighboring members. Then one may also add negative numbers represented in the complementary code, where a negative n-bit number is represented as $-x = 2^n - x$, so that the value of a bit string $A = a_0 \dots a_n$ is calculated as $\text{value}(A) = -a_n \cdot 2^n + \sum_{i=0}^{n-1} a_i \cdot 2^i$. By Abel's trick, we may multiply two numbers $X = x_0 x_1 \dots x_{N-1}$ and $Y = y_0 y_1 \dots y_{N-1}$ by using the following relation:

$$X \cdot Y = -X y_0 + \sum_{i=1}^{N-1} X(y_{i-1} - y_i) 2^i + X y_{N-1} 2^N.$$

If two neighboring bits of Y are equal, we may step over one of the steps, as we did in modified add-step. This scheme on average works faster than a regular add-stepper algorithm; in its worst-case situation, where all neighboring bits are different (010101 ... or 101010 ...) it runs for just the same time as an add-step multiplier.

Formally, this leads us to the following algorithm:

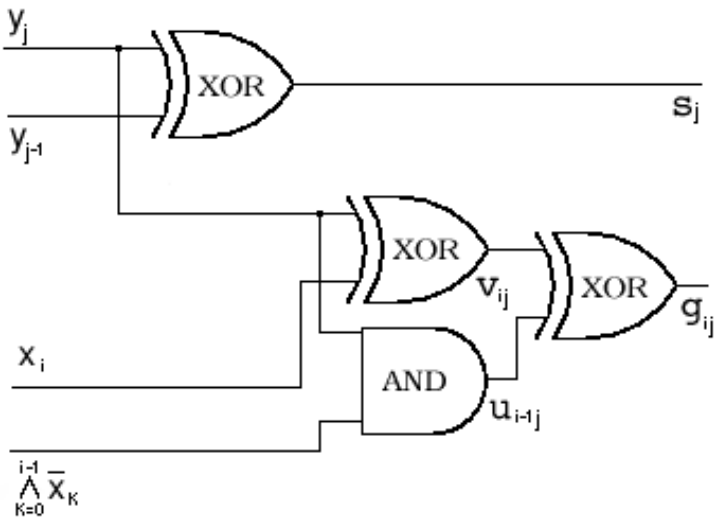

Figure A.5: Part of the cell for the booth multiplier.

- **Input:** $X = x_0 \dots x_N$, $Y = y_0 \dots y_M$
- **Algorithm:** Calculate every member of the sum one by one (we set y_{-1} to be 0)

$$\begin{aligned} X \cdot (y_{-1} - y_0) \cdot 2^0 + \\ X \cdot (y_0 - y_1) \cdot 2^1 + \\ \dots + \\ X \cdot (y_{M-1} - y_M) \cdot 2^M = \\ X \cdot (-y_M \cdot 2^M + \sum_{i=0}^{M-1} y_i \cdot 2^i) = \\ X \cdot \text{value}(Y) \end{aligned}$$

- **Output:** $C = X \cdot \text{value}(Y)$

As a boolean circuit acting as a Booth multiplier we take the following. We consider the circuit from the previous section as the basic structure for the Booth circuit and substitute x_i and y_j for the schemes depicted on Fig. A.5.

However, before proceeding to the Booth multiplier, we would have to face a simpler challenge: a multiplier which we called *simplified Booth*.

Simplified Booth multiplier differs from the original Booth multiplier in that it does not employ the "modified" trick (which we described in the beginning of this section). The original Booth multiplier may be obtained from the simplified Booth multiplier in the same way as we obtained modified add-stepper from the original add-stepper multiplier.

This simplified circuit works in fact even slower than the original add-stepper algorithm (all its power is driven away by not stepping over zero computations, and the constant overhead is larger). However, it is a necessary step on our way to the original Booth multipliers, as a proof of equivalence between simplified Booth and add-stepper would immediately (although somewhat messy, very technical and at present infeasible for the solver, so we do not write it down here) yield an equivalence proof between original Booth and modified add-step.

We now write down the clauses which constitute the boolean equivalent of the simplified Booth circuit. Our description is divided into several part with respect to the function of different clauses in the formula.

A circuit for representing negative numbers:

We add the following sets of clauses:

1. Clauses describing s_j, $1 \leq j \leq M$:

$$\left.\begin{aligned} \neg s_0 \vee y_0 \\ s_0 \vee \neg y_0 \end{aligned}\right\} \quad s_0 = y_0$$

$$\left.\begin{aligned} \neg y_j \vee y_{j-1} \vee s_j \\ y_j \vee \neg y_{j-1} \vee s_j \\ y_j \vee y_{j-1} \vee \neg s_j \\ \neg y_j \vee \neg y_{j-1} \vee \neg r_j \end{aligned}\right\} \quad s_j = y_j \oplus y_{j-1}$$

$$\left.\begin{aligned} \neg s_{M+1} \vee y_M \\ s_{M+1} \vee \neg y_M \end{aligned}\right\} \quad s_{M+1} = y_M$$

2. Clauses describing negations of x_i, $0 \leq i \leq N$:

$$\left.\begin{aligned} nx_i \vee x_i \\ \neg nx_i \vee \neg x_i \end{aligned}\right\} \quad nx_i = \neg x_i$$

3. Clauses describing $\wedge_{K=0}^{i} \neg x_K$, $1 \leq i \leq N$:

$$\left.\begin{aligned} \neg w_0 \vee nx_0 \\ w_0 \vee \neg nx_0 \end{aligned}\right\} \quad w_0 = nx_0$$

$$\left.\begin{aligned} w_{i-1} \vee \neg w_i \\ nx_i \vee \neg w_i \\ w_i \vee \neg w_{i-1} \vee \neg nx_i \end{aligned}\right\} \quad w_i = w_{i-1} \wedge nx_i$$

4. Clauses defining variables g_{ij}, $1 \leq i \leq N$, $0 \leq j \leq M$:

$$\left.\begin{aligned} \neg g_{0j} \vee x_0 \\ g_{0j} \vee \neg x_0 \end{aligned}\right\} \quad g_{0j} = x_0$$

$$\left.\begin{aligned} \neg y_j \vee x_i \vee v_{ij} \\ y_j \vee \neg x_i \vee v_{ij} \\ y_j \vee x_i \vee \neg v_{ij} \\ \neg y_j \vee \neg x_i \vee \neg v_{ij} \end{aligned}\right\} \quad v_{ij} = y_j \oplus x_i$$

$$\left.\begin{aligned} w_{i-1} \vee \neg u_{i-1j} \\ y_j \vee \neg u_{i-1j} \\ u_{i-1j} \vee \neg w_{i-1} \vee \neg y_j \end{aligned}\right\} \quad u_{i-1j} = w_{i-1} \wedge y_j$$

$$\left.\begin{aligned} \neg v_{ij} \vee u_{i-1j} \vee g_{ij} \\ v_{ij} \vee \neg u_{i-1j} \vee g_{ij} \\ v_{ij} \vee u_{i-1j} \vee \neg g_{ij} \\ \neg v_{ij} \vee \neg u_{i-1j} \vee \neg g_{ij} \end{aligned}\right\} \quad g_{ij} = v_{ij} \oplus u_{i-1j}$$

$$\left.\begin{aligned} \neg g_{0M+1} \vee x_0 \\ g_{0M+1} \vee \neg x_0 \end{aligned}\right\} \quad g_{0M+1} = x_0$$

$$\left.\begin{array}{r} \neg g_{iM+1} \vee x_i \\ g_{iM+1} \vee \neg x_i \end{array}\right\} \qquad g_{iM+1} = x_i$$

A circuit for a simple add-stepper multiplier[2]:

5. The zeroth column is defined by variables r_{i0}, $0 \leq i \leq N$:

$$\left.\begin{array}{r} g_{i0} \vee \neg r_{i0} \\ s_0 \vee \neg r_{i0} \\ r_{00} \vee \neg g_{i0} \vee \neg s_0 \end{array}\right\} \qquad r_{i0} = g_{i0} \wedge s_0$$

$$\left.\begin{array}{r} nw_N \vee w_N \\ \neg nw_N \vee \neg x_N \end{array}\right\} \qquad nw_N = \neg w_N$$

$$\left.\begin{array}{r} nw_N \vee \neg r_{N0} \\ y_0 \vee \neg r_{N0} \\ r_{N0} \vee \neg nw_N \vee \neg y_0 \end{array}\right\} \qquad r_{N0} = nw_N \wedge y_0$$

6. The lowest cell in each of the following columns ($j = 1..M+1$) is defined as follows:

$$\left.\begin{array}{r} g_{0j} \vee \neg c_{0j} \\ s_j \vee \neg c_{0j} \\ c_{0j} \vee \neg g_{0j} \vee \neg s_j \end{array}\right\} \qquad c_{0j} = g_{0j} \wedge s_j$$

$$\left.\begin{array}{r} \neg c_{0j} \vee r_{1j-1} \vee r_{0j} \\ c_{0j} \vee \neg r_{1j-1} \vee r_{0j} \\ c_{0j} \vee r_{1j-1} \vee \neg r_{0j} \\ \neg c_{0j} \vee \neg r_{1j-1} \vee \neg r_{0j} \end{array}\right\} \qquad r_{0j} = c_{0j} \oplus r_{1j-1}$$

$$\left.\begin{array}{r} c_{0j} \vee \neg t_{0j} \\ r_{1j-1} \vee \neg t_{0j} \\ t_{0j} \vee \neg c_{0j} \vee \neg r_{1j-1} \end{array}\right\} \qquad t_{0j} = c_{0j} \wedge r_{1j-1}$$

7. Upper cells in the remaning columns ($j = 1..M+1$) are defined by the following clauses:

$$\left.\begin{array}{r} nw_N \vee \neg k_{N+1j} \\ y_j \vee \neg k_{N+1j} \\ k_{N+1j} \vee \neg nw_N \vee \neg y_j \end{array}\right\} \qquad k_{N+1j} = nw_N \wedge y_j$$

$$\left.\begin{array}{r} k_{N+1j} \vee \neg c_{N+1j} \\ s_j \vee \neg c_{N+1j} \\ c_{N+1j} \vee \neg k_{N+1j} \vee \neg s_j \end{array}\right\} \qquad c_{N+1j} = k_{N+1j} \wedge s_j$$

$$\left.\begin{array}{r} \neg t_{N+1j} \vee c_{N+1j} \vee r_{N+1j-1} \\ t_{N+1j} \vee \neg c_{N+1j} \vee r_{N+1j-1} \\ t_{N+1j} \vee c_{N+1j} \vee \neg r_{N+1j-1} \\ \neg t_{N+1j} \vee \neg c_{N+1j} \vee \neg r_{N+1j-1} \end{array}\right\} \qquad t_{N+1j} = c_{N+1j} \oplus r_{N+1j-1}$$

[2] in the simplified Booth its purpose is to add up $g_i \cdot 2^i$ introduced in the previous item

$$\left.\begin{array}{r} \neg t_{N+1j} \vee t_{Nj} \vee r_{N+1j} \\ t_{N+1j} \vee \neg t_{Nj} \vee r_{N+1j} \\ t_{N+1j} \vee t_{Nj} \vee \neg r_{N+1j} \\ \neg t_{N+1j} \vee \neg t_{Nj} \vee \neg r_{Nj} \end{array}\right\} \quad r_{N+1j} = t_{N+1j} \oplus t_{Nj}$$

8. The rest of the cells (the rest of the set $\{i,j\}_{i=0..N,j=0..M+1}$) is defined by the following sets of clauses:

$$\left.\begin{array}{r} g_{ij} \vee \neg c_{ij} \\ s_j \vee \neg c_{ij} \\ c_{ij} \vee \neg g_{ij} \vee \neg s_j \end{array}\right\} \quad c_{ij} = g_{ij} \wedge s_j$$

$$\left.\begin{array}{r} \neg c_{ij} \vee r_{i+1,j-1} \vee k_{ij} \\ c_{ij} \vee \neg r_{i+1,j-1} \vee k_{ij} \\ c_{ij} \vee r_{i+1,j-1} \vee \neg k_{ij} \\ \neg c_{ij} \vee \neg r_{i+1,j-1} \vee \neg k_{ij} \end{array}\right\} \quad k_{ij} = c_{ij} \oplus r_{i+1,j-1}$$

$$\left.\begin{array}{r} \neg k_{ij} \vee t_{i-1,j} \vee r_{ij} \\ k_{ij} \vee \neg t_{i-1,j} \vee r_{ij} \\ k_{ij} \vee t_{i-1,j} \vee \neg r_{ij} \\ \neg k_{ij} \vee \neg t_{i-1,j} \vee \neg r_{ij} \end{array}\right\} \quad r_{ij} = k_{ij} \oplus t_{i-1,j}$$

$$\left.\begin{array}{r} c_{ij} \vee \neg l_{ij} \\ r_{i+1,j-1} \vee \neg l_{ij} \\ l_{ij} \vee \neg c_{ij} \vee \neg r_{i+1,j-1} \end{array}\right\} \quad l_{ij} = c_{ij} \wedge r_{i+1,j-1}$$

$$\left.\begin{array}{r} c_{ij} \vee \neg m_{ij} \\ t_{i-1,j} \vee \neg m_{ij} \\ m_{ij} \vee \neg c_{ij} \vee \neg t_{i-1,j} \end{array}\right\} \quad m_{ij} = c_{ij} \wedge t_{i-1,j}$$

$$\left.\begin{array}{r} r_{i+1,j-1} \vee \neg n_{ij} \\ t_{i-1,j} \vee \neg n_{ij} \\ n_{ij} \vee \neg r_{i+1,j-1} \vee \neg t_{i-1,j} \end{array}\right\} \quad n_{ij} = r_{i+1,j-1} \wedge t_{i-1,j}$$

$$\left.\begin{array}{r} \neg l_{ij} \vee e_{ij} \\ \neg m_{ij} \vee e_{ij} \\ \neg e_{ij} \vee l_{ij} \vee m_{ij} \end{array}\right\} \quad e_{ij} = l_{ij} \vee m_{ij}$$

$$\left.\begin{array}{r} \neg n_{ij} \vee t_{ij} \\ \neg e_{ij} \vee t_{ij} \\ \neg t_{ij} \vee n_{ij} \vee e_{ij} \end{array}\right\} \quad t_{ij} = n_{ij} \vee e_{ij}$$

9. $r_{0j}, 0 \leq j \leq M+1$ correspond to the first $M+2$ outputs of the simplified Booth multilplier, $p_i, 0 \leq i \leq M+1$. $r_{iM+1}, 1 \leq i \leq N$ corresponding to the next N outputs of the circuit.

10. We treat the case where one of the factors is a one-bit number as a special one and do not consider it here.