Scalable Design and Synthesis of Reversible Circuits

Eleonora Schönborn

A dissertation submitted for the degree of
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

In the FB3 – Mathematik und Informatik
Universität Bremen

Primary supervisor: Prof. Dr. Rolf Drechsler
Secondary supervisor: Prof. Dr. Gerhard Dueck

Date of the doctoral colloquium: September 21, 2016

Acknowledgements

First and foremost I want to thank my supervisor Prof. Dr. Rolf Drechsler for giving me the opportunity to research this topic, for always supporting my ideas, and giving me the right amounts of guidance and freedom in my work. He saw my potential before I did, and I greatly value the trust he put in me.

I am deeply grateful to Prof. Dr. Robert Wille, who greatly inspired me with his enthusiasm. Without his expert knowledge on reversible logic, all our fruitful discussions, and his support and encouragement, I could never have written this thesis.

My heartfelt appreciation goes to Dr. Mathias Soeken, who despite his high workload always found time for me. His expert knowledge, valuable feedback, and constant support were an essential contribution to this dissertation. In particular, he showed a lot of patience in mediating between RevKit and my computer.

Especially, I want to thank Professor Gerhard Dueck for his willingness to review my thesis.

Another aspect that greatly added to this dissertation is the welcoming and productive working environment I found in the Group of Computer Architecture at the University of Bremen. I would like to thank everyone in this group, and in the Graduate School System Design (SyDe), for their contribution. Particularly helpful for my work has been the input from Dr. Michael Kirkedal Thomsen and Nils Przigoda.
For being outstandingly helpful, supportive, and awesome, I would like to thank Dr. Julia Seiter, Dr. Melanie Diepenbeck, and Jannis Stoppe.

My special thanks go to Dr. Kamalika Datta, Professor Indranil Sengupta, and Professor Hafizur Rahaman for the productive cooperation and great experiences.

Last but not least, I want to thank my family and my friends for everything, especially their support during the recent years.

Contents

List of Figures

List of Tables

1. Introduction

Computational components are being embedded in more and more objects of our everyday lives. In smartphones, cars, medical equipment, etc. these components are linked closely to their physical environment using sensors and actors. Connected via networks they form cyber-physical systems. The expectations on these integrated circuits are rising with their number of applications. Especially low energy consumption has become a crucial design goal. While established power management techniques are reaching their limits, technologies alternative to CMOS are becoming more important day by day.

Many alternative technologies and applications currently investigated are based on reversible computation, a computing paradigm which only allows reversible operations. Examples include applications in the domain of

- *Encoding and Decoding Devices*, which always realize one-to-one mappings and, thus, inherently follow a reversible computing paradigm (see e.g. [WDOGO12]),
- *Quantum Computation*, which enables to solve many relevant problems significantly faster than conventional circuits and inherently is reversible (see e.g. [NC00]),
- *Low Power Computation*, where the fact that no information is lost in reversible computation may be exploited in the future (see e.g. [Lan61, BAP+12]),
- *Adiabatic Circuits*, a special low power technology that reversible circuits are particularly suited for (see e.g. [PF96]), and
- *Program Inversion* (see e.g. [GK05]), as programs based on a reversible computation paradigm would allow an inherent and obvious program inversion.

While some of these applications are still in a prototypical stage, impressive improvements have been made in the recent years, e.g. more scalable quantum circuits [VSB+01] or an experimental validation of the low power properties of reversible computation [BAP+12]. In contrast, the development of proper design methods for this kind of circuits seems to still be in its infancy.

For conventional circuits, an elaborated design flow emerged over the last 20-30 years. A hierarchical flow composed of several abstraction levels (e.g. the formal specification level, the electronic system level, the register transfer level, and the gate level) and supported by a wide range of modeling languages, system description languages, and hardware description languages (HDLs) has been developed and is in industrial use. While mainly relying on this *conventional* way of computation, elaborated design flows for alternative computing paradigms seem to remain in the distant future.

Since reversible computation only allows reversible, i.e. bijective, operations, each gate in a reversible circuit represents a bijection. Conventional gate libraries can not be applied here, and new libraries of reversible gates have been introduced. Furthermore, fanout and feedback are generally not allowed in reversible circuits. As a consequence, design methods can not simply be transferred from conventional circuit design, but have to be adapted or developed from scratch.

Essential features and approaches of modern design flows are not available to reversible circuit design yet. Most existing approaches work on the gate level, i.e. almost no support for reversible circuits and systems on the specification level, the electronic system level, or the register transfer level exists yet. Moreover, most of the existing approaches for synthesis only accept specifications provided in terms of Boolean function descriptions like truth tables or Boolean decision diagrams (see e.g. [SM11]). Only very preliminary hardware description languages are available thus far [WOD10, Tho12]. Hence, after more than a decade of research in the design of reversible circuits, there is hardly an answer for how to scale the design capabilities for reversible circuits.

In this thesis, we investigate scalable approaches to the design and synthesis of reversible circuits. Two complementary directions are discussed, namely (1) designing reversible circuits by exploiting the conventional design flow first, and afterwards mapping the result to a reversible circuit, and (2) applying an entirely new design flow to be developed, which considers reversibility right from the beginning through all abstraction levels.

Exploiting the Conventional Design Flow

The design flow for conventional circuits has been continually improved over decades and offers many powerful design tools and algorithms. Here, we consider using these methods for the design of reversible circuits. To be precise, the first steps of the design process follow the conventional design flow. The resulting conventional design will then automatically be mapped to a reversible circuit description.

When following this direction, the most important questions are:

- At which abstraction level should the conventional design be mapped to a reversible circuit description?
- How can the mapping be done efficiently with regards to runtime as well as the resulting circuit design?

Mapping at a low abstraction level like the gate level can be realized straightforwardly. Each conventional gate is substituted by a template of reversible gates realizing the same function or, in the case of irreversible functions, embedding the function in a bijection using additional circuit lines. However, since each gate is mapped individually without regarding global information, the resulting circuits are usually far from optimal.

In Chapter 4 we consider an approach mapping from the register transfer level instead. The mapping scheme is similar to the one described for the gate level, but instead of single gates, complete modules have to be substituted. For this purpose, past accomplishments

in the design of reversible building blocks for various data flow operations like adders, multipliers, etc. can be exploited. This way, circuit lines and/or gate cost can be saved compared to the gate level mapping.

Mapping from a higher level of abstraction, like the HDL description, would enable the use of even more global information and thus further reductions in the resulting circuits. However, this would require a complex mapping scheme yet to be developed.

Developing a Specific Design Flow

The second direction aims for the development of an entirely new design flow which considers reversibility from the specification and through all following abstraction levels. Special characteristics of reversible functions could be exploited this way. Theoretically, there would be no need for embedding. On the downside, the whole design flow has to be redeveloped.

For the specification of large and/or complex reversible systems, HDLs supporting the characteristics of reversible logic have to be developed. Thus far, only preliminary versions of such HDLs are available (e.g. [WOD10, Tho12]).

However, it is already possible to synthesize a reversible circuit directly from the HDL description, e.g. with an algorithm we review in Chapter 6. Here, a statement like `c^=a*b` is realized by cascading building blocks for the operations (multiplication and XOR-assignment). Since non-reversible parts of the overall reversible statement are synthesized separately, additional circuit lines are required for embedding. Hence, this synthesis scheme suffers from similar problems as the mapping methods discussed in Chapter 4. But in contrast, the initial reversible description allows for un-computing temporary results and thus for saving some of the additional lines, as we show in Chapter 8.

There are some grave differences between these reversible HDLs and the conventional ones. For example, direct assignments such as `a=b` are not allowed because of their irreversibility. Despite these differences, those languages enable the design of complex systems in reversible logic as we show in Chapter 9.

These conceptual differences also exist between reversible and conventional circuits. While circuit designers have gained an intuitive knowledge about conventional circuits and their properties, such an intuition has yet to be acquired in the reversible domain. To this end, we developed RevVis, the first tool for visualizing structures and properties of reversible circuits, as introduced in Chapter 10. This visualization might inspire new ideas regarding synthesis, optimization, or debugging.

To efficiently design reversible logic, we need to investigate high abstraction levels like HDL. In this work, two directions are considered: Exploiting the conventional design flow and developing a new flow according to the properties of reversible circuits. Which direction should be taken is not obvious and may depend on the application. Thus, we discuss the possible assets and drawbacks of taking either direction. We present ideas which can be exploited and outline open challenges which still have to be addressed. Preliminary results obtained by initial implementations illustrate the way to go. By

this we present and discuss two promising and complementary directions for the scalable design and synthesis of reversible circuits.

The thesis is structured as follows.

Chapter 2 – Reversible Logic

To keep this document self-contained, preliminaries are provided in this chapter. These include the basics of reversible circuits and the cost metrics used in this work.

Chapter 3 – Using Conventional Data Structures

An algorithm is reviewed which is based on decision diagrams, a data structure used to represent conventional circuits, and maps them to reversible circuits. We propose and compare two different optimizations for this synthesis algorithm. Both are employing negative control lines to reduce the gate count and gate cost.

Chapter 4 – Mapping Conventional Circuits

In this chapter, we discuss the direct mapping of conventional to reversible circuits. A mapping from the register transfer level is developed to provide more scalability and efficiency compared to a gate level mapping. This approach is compared to the synthesis of reversible circuits from reversible specifications as described in Chapter 6.

Chapter 5 – The SyReC Language

SyReC (first introduced in [WOD10]) is the reversible HDL we chose to use in developing a specific design flow for reversible circuits. This chapter introduces the general concepts, syntax and semantics of the language in its recent form.

Chapter 6 – SyReC Synthesis

Here, the synthesis algorithm is reviewed which maps a SyReC specification to a reversible circuit.

Chapter 7 – SyReC Building Blocks

A building block in SyReC determines how an operation (e.g. assignment, addition) is mapped to reversible gates. This chapter is the first document to explain each of these mappings in detail. Some of the building blocks have been improved in the process of this work, yet minimality is not guaranteed.

Chapter 8 – Optimization of SyReC Synthesis

We propose an extended synthesis scheme for SyReC specifications to reduce the number of resulting circuit signals. While the number of signals can be strikingly decreased, this optimization comes at the cost of additional gates, resulting in a trade-off the designer should decide on. Additionally, we employ an existing method to reduce the gate cost, and evaluate the separate and combined effects of both optimizations.

Chapter 9 – Designing a RISC CPU in Reversible Logic

In this chapter, the applicability of a reversible design flow is tested. Given a textual specification of a conventional RISC CPU, we identify the components and design the computational parts of the CPU in reversible logic, using the SyReC language. The functionality is tested by simulating the execution of a software program on the proposed CPU.

Chapter 10 – Visualization of Structures and Properties of Reversible Circuits

Reversible circuits are usually visualized by simple netlist representations. We propose the first visualization to highlight structures and properties of reversible circuits, which is especially useful for large circuits. With this, an intuition for this kind of circuits might be acquired and help to develop and improve design, synthesis, verification, testing methods etc. We compare the structures and properties of circuits generated with different synthesis approaches.

Chapter 11 – Conclusion

In the final chapter, the contents of this work are summarized.

The main ideas in this thesis have already been or will be published in the following articles.

- General Idea, Chapter 4:

  E. Schönborn, R.Wille, and R. Drechsler. Quo Vadis, Reversible Circuit Design? Towards Scaling Design and Synthesis of Reversible Circuits. In *Reed-Muller Workshop*, 2015.

- Chapter 3:

  E. Schönborn, K. Datta, R. Wille, I. Sengupta, H. Rahaman, and R. Drechsler. Optimizing DD-based Synthesis of Reversible Circuits using Negative Control Lines. In *IEEE Int'l Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 129–134, 2014.

- Chapter 5,6:

  R. Wille, E. Schönborn, M. Soeken, and R. Drechsler. SyReC: A Hardware Description Language for the Specification and Synthesis of Reversible Circuits. *Integration, the VLSI Journal*. In press.

- Chapter 8:

  R. Wille, M. Soeken, E. Schönborn, and R. Drechsler. Circuit Line Minimization in the HDL-Based Synthesis of Reversible Logic. In *IEEE Annual Symposium on VLSI*, pages 213–218, 2012.

- Chapter 9:

  R. Wille, M. Soeken, D. Große, E. Schönborn, and R. Drechsler. Designing a RISC CPU in Reversible Logic. In *Int'l Symposium on Multi-Valued Logic*, pages 170–175, 2011.

- Chapter 10:

  R. Wille, J. Stoppe, E. Schönborn, K. Datta, and R. Drechsler. RevVis: Visualization of Structures and Properties in Reversible Circuits. In *Reversible Logic*, pages 111–124, 2014.

2. Reversible Logic

To keep the thesis self-contained, the preliminaries are given in this chapter. Note that only brief introductions of the concepts and notations are given. For further reading, please consult the references given in the corresponding sections.

This chapter is divided into two parts. First, the definition of reversible functions is given, and the concept of embedding is introduced. Next, reversible circuits as used in this work are defined, and the metrics for measuring their cost are given.

2.1. Reversible Functions

A propositional or Boolean function $f : \mathbb{B}^n \to \mathbb{B}^n$ over the *variables* $X = \{x_1, \ldots, x_n\}$ is called *reversible* if it is bijective. Clearly, many Boolean functions of practical interest are not reversible. These include bitwise conjunction, disjunction, binary addition, and multiplication of two bit strings. In order to realize such functionality in a reversible circuit, the corresponding functions are *embedded* [MD04a, WKD11].

To embed a non-reversible function f, a reversible function f' is constructed, so that f' contains the function f. This is achieved by adding so-called *garbage outputs* to f which are used to distinguish equal output patterns, thus making the function injective. If necessary, *constant inputs* are added to equalize the number of input variables and output variables of the function, thus making it bijective. These inputs are called constant as f' is only defined to behave like f if a certain value (i.e. 0 or 1) is constantly present at those inputs.

Example 1 *Table 2.1a shows the truth table for the conjunction. The value of f is 1, iff both x_1 and x_2 are 1. It is easy to see that this function is not reversible: If f has the value 0, the value of the inputs can not be concluded.*

In Table 2.1b, garbage outputs were added to differentiate the identical output patterns of f. Since f has the same output for three different input patterns, two garbage outputs need to be added to distinguish all cases. To make the function bijective, the number of inputs and outputs has to be identical, so a constant input is added.

Table 2.1c shows a reversible function embedding the conjunction. The original function f is highlighted in grey. The values outside the scope of f can be chosen freely, as long as the whole truth table represents a bijection.

Table 2.1.: Embedding the Conjunction

(a) Conjunction

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

(b) Irreversible Function

x_1	x_2	f	g_1	g_2
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

(c) Reversible Function

0	x_1	x_2	f'	g_1	g_2
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

2.2. Reversible Circuits

Reversible functions can be realized by reversible circuits in which each variable of the function is represented by a *circuit line*. To maintain the bijectivity property of the reversible function, fan-out and feedback are not directly allowed in reversible circuits. As a consequence, reversible circuits can be built as a cascade of reversible gates $G = g_1 \dots g_d$. There exist different gate libraries that are being used to build reversible circuits. However, in the scope of this work we restrict ourselves to the most commonly used ones containing the *Toffoli gate* [Tof80] and the *Fredkin gate* [FT82]. For this purpose each gate g_i in the circuit is denoted by $t(C, T)$ with

- a gate type $t \in \{\mathrm{T}, \mathrm{F}\}$,
- *control lines* $C \subset X$, and
- *target lines* $T \subseteq X \setminus C$.

Each gate g_i realizes a reversible function $f_i : \mathbb{B}^n \to \mathbb{B}^n$. If $t = \mathrm{T}$, i.e. the gate is a Toffoli gate, we have $T = \{x_t\}$ and f_i maps

$$(x_1, \dots, x_n) \mapsto (x_1, \dots, x_{t-1}, x_t \oplus \bigwedge_{c \in C} c, x_{t+1}, \dots, x_n),$$

i.e. the value on line x_t is inverted if and only if all control values are assigned 1. A Toffoli gate is called a NOT gate if $|C| = 0$. For a Fredkin gate, i.e. $t = \mathrm{F}$, we have $T = \{x_s, x_t\}$ and f_i maps

$$\begin{aligned}(x_1, \dots, x_n) \mapsto \\ (x_1, \dots, x_{s-1}, x'_s, x_{s+1}, \dots, x_{t-1}, x'_t, x_{t+1}, \dots, x_n),\end{aligned}$$

with $x'_s = \bar{c}' x_s \oplus c' x_t$, $x'_t = \bar{c}' x_t \oplus c' x_s$, and $c' = \bigwedge_{c \in C} c$, i.e. the values of the target lines are interchanged (swapped) if and only if all control values are assigned 1. A Fredkin gate is also referred to as SWAP gate if $|C| = 0$. The function realized by the circuit is the composition of the functions realized by the gates, i.e. $f = f_1 \circ f_2 \circ \dots \circ f_d$.

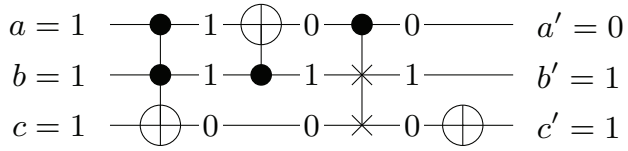


Figure 2.1.: Example of a Reversible Circuit

Example 2 *Fig. 2.1 shows a reversible circuit with three lines and four gates. The first, second, and fourth gates are Toffoli gates with a different number of control lines. The target line is denoted by* $\oplus$ *whereas the control lines are denoted as solid black dots. The third gate is a Fredkin gate which target lines are denoted by* $\times$*.*

In Chapter 3 and 10, we additionally consider Toffoli and Fredkin gates with *negative control lines*. A gate can have both positive and negative control lines. In this case, the respective functionality is applied to the target line(s) if and only if all values on positive controls are assigned 1 and all values on negative controls are assigned 0. Negative control lines are depicted as an empty dot.

In addition to the constant inputs and garbage outputs that are added to a function in the process of embedding, for circuits we are also considering so-called *ancilla lines*. Ancilla lines hold a constant input assigned some Boolean value v and are used in such a way that their output is always v. Moreover, when considering circuits that realize a complex functionality some lines may be semantically grouped as a *signal*, e.g. if the circuit realizes the addition of two 32-bit values.

In order to measure the costs of a circuit, different metrics are being applied. Besides the number of gates, so-called *quantum costs* and *transistor costs* approximate a better cost considering the actual physical implementation based on quantum mechanics and classical mechanics, respectively. Most of the cost metrics are applied to the gates and are accumulated in order to calculate the costs for the overall circuit.

In this work, we are using the quantum cost metrics presented in [BBC+95] with the optimizations from [MD04b] and [MYMD05]. Table 2.2 shows the quantum cost for a selection of Toffoli and Fredkin gates. *Free lines* refer to lines that are present in the circuit, but not connected to the respective gate, i.e. neither control nor target lines. As can be seen, the quantum cost of a gate can grow exponentially with respect to the number of control lines.

The transistor cost TC estimate the effort needed to realize a reversible gate in CMOS according to [DV02]. A reversible gate with s control lines has a transistor cost of $8 \cdot s$, which is shown for the various examples in Table 2.2.

Table 2.2.: Cost Metrics for Toffoli and Fredkin Gates

#Control Lines	Quantum Cost		TC
	Toffoli Gate	Fredkin Gate	
0	1	3	0
1	1	7	8
2	5	15	16
3	13	28, if at least two lines are free 31, otherwise	24
4	26, if at least two lines are free 29, otherwise	40, if at least three lines are free 54, if one or two lines are free 63, otherwise	32
5	38, if at least three lines are free 52, if one or two lines are free 61, otherwise	52, if at least four lines are free 82, if one to three lines are free 127, otherwise	40
6	50, if at least four lines are free 80, if one to three lines are free 125, otherwise	64, if at least five lines are free 102, if one to four lines are free 255, otherwise	48
7	62, if at least five lines are free 100, if one to four lines are free 253, otherwise	76, if at least six lines are free 130, if one to five lines are free 511, otherwise	56

Part I.

Exploiting the Conventional Design Flow

3. Using Conventional Data Structures

Reversible circuits are constructed by creating a cascade of basic reversible gates, like NOT, controlled NOT [Fey85], or Toffoli gates [Tof80], with additional constraints like no direct support of fanout and feedback. Because of these constraints as well as the new gate library, synthesis of reversible circuits significantly differs from the design of conventional circuits.

Consequently, new approaches for the synthesis of reversible circuits have been explored by researchers. These include

- exact methods [GWDD09] for obtaining optimal circuits, which, due to their computational complexity, work for very small functions only,
- constructive approaches [MDM05, GAJ06] which are able to synthesize relatively large functions (with up to 30 inputs), and
- methods based on *Decision Diagrams* (DDs, [WD09]) or *Exclusive Sum of Products* (ESOPs, [FTR07]) which enable synthesis for very large functions.

In these methods, the given function to be synthesized is represented using different function descriptions such as truth tables, DDs, or ESOPs. In the following, we focus on DD-based synthesis. Here, a hierarchical approach is applied which uses a DD to represent the function to be synthesized and transforms each node into a corresponding sub-circuit (this approach is reviewed in more detail later in Section 3.1). Thus far, all existing approaches following this scheme (such as [WD09, SWD10, WD10]) rely on a gate library composed of Toffoli gates with positive control lines only. Recently, an extension of these gates with mixed control lines, i.e. with both positive and negative control lines, received attention. It has been shown that additionally considering negative control lines enables the synthesis of reversible circuits with significantly less costs [WSPD12, DSR13, DRW$^+$13, ST13]. However, these recent findings have not yet been exploited for DD-based synthesis.

In this work, we investigate the potential of utilizing negative control lines for DD-based synthesis. To this end, we consider

- how the application of an existing (post-synthesis) optimization approach utilizing negative control lines improves the circuit realizations obtained by DD-based synthesis, and
- how negative control lines can explicitly be exploited during the synthesis.

Both schemes have been evaluated. The results clearly show that the utilization of negative control lines significantly reduces the costs of the respective circuits. In the best cases, up to 43% of the gates and 15% of the quantum costs can be saved.

The remainder of this chapter is organized as follows. Section 3.1 briefly reviews DD-based synthesis and motivates this work. Afterwards, the explicit consideration of negative control lines in DD-based synthesis and the post-synthesis optimization scheme are discussed in Section 3.2 and Section 3.3, respectively. Finally, Section 3.4 summarizes the experimental evaluation and Section 3.5 concludes the chapter.

3.1. DD-based Synthesis

DD-based synthesis is a hierarchical synthesis approach which enables the automatic generation of a reversible circuit realizing a given function f. To this end, the function f to be synthesized is decomposed into smaller sub-functions. This decomposition is repeatedly applied until the sub-functions evaluate to a constant 0 or 1. By this, the (possibly very large) function f is represented by a logical combination of co-factors. While the overall function f is usually hard to synthesize in one step, the respective co-factors as well as logical combinations resulting from the decomposition are rather small and, hence, can easily be realized as sub-circuits. Composing all these sub-circuits eventually results in a circuit realizing the desired function f.

This scheme has originally been applied in [WD09] and further refined e.g. in [SWD10, WD10]. The decompositions have been conducted by the use of data structures like *Binary Decision Diagrams* (BDDs, [Bry86]) or *Kronecker Functional Decision Diagrams* (KFDDs, [DST+94, DB06]). Both are directed, acyclic graphs $G = (V, E)$ with a root that represents the function f. Each inner node $v \in V$ has two child nodes low(v) and high(v) representing the sub-functions obtained by the decomposition. Possible decompositions are defined by:

$$f = \overline{x}_i \cdot f_{x_i=0} + x_i \cdot f_{x_i=1} \qquad \text{(Shannon)}$$
$$f = f_{x_i=0} \oplus x_i \cdot (f_{x_i=0} \oplus f_{x_i=1}) \qquad \text{(positive Davio)}$$
$$f = f_{x_i=1} \oplus \overline{x}_i \cdot (f_{x_i=0} \oplus f_{x_i=1}) \qquad \text{(negative Davio)}$$

Each inner node is labeled with a variable of f and each variable is assigned a decomposition type. For example, if a node representing the function f is labeled with the variable x_i which is assigned Shannon decomposition, its child nodes represent $f_{x_i=0}$ (low(v)) and $f_{x_i=1}$ (high(v)), where $f_{x_i=0}$ ($f_{x_i=1}$) is the negative (positive) co-factor of f obtained by assigning x_i to 0 (1). Co-factors evaluating to the constant 0 or 1 are represented by terminal nodes.

Note that BDDs only allow Shannon decomposition, while KFDDs support all decompositions mentioned above. In this sense, BDDs are a special case of KFDDs. Due to the reduced diagram complexity, algorithms for BDDs are often more efficient than those for KFDDs. On the other hand, KFDDs allow for a more compact representation of certain types of Boolean functions [BDW95]. In the following, we generically denote these data structures by *Decision Diagrams* (DDs).

For an even more compact representation of functions in a DD, *complement edges* have been introduced. If a complement edge is pointing to a node v, the function $\overline{f_v}$ rather than the function f_v is used. These edges are denoted by a $\bullet$ in the following figures.

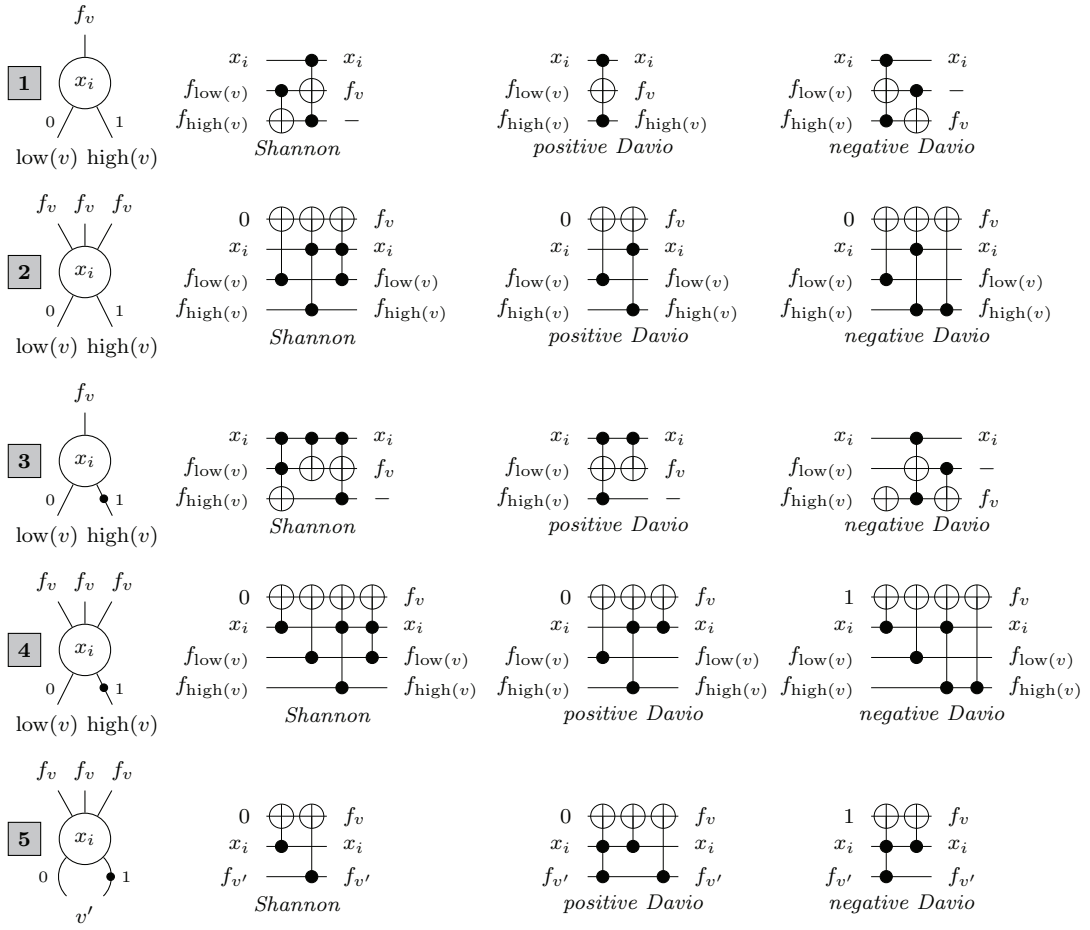


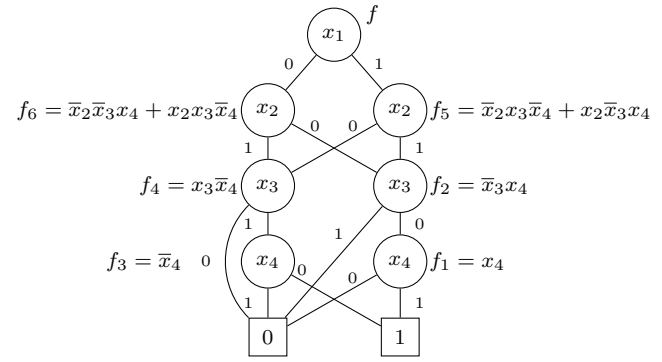
Figure 3.1.: Reversible Cascades Representing the Different DD Decompositions

Taking all that into consideration, synthesis for a given function f represented by a DD $G = (V, E)$ can be performed by conducting the following steps:

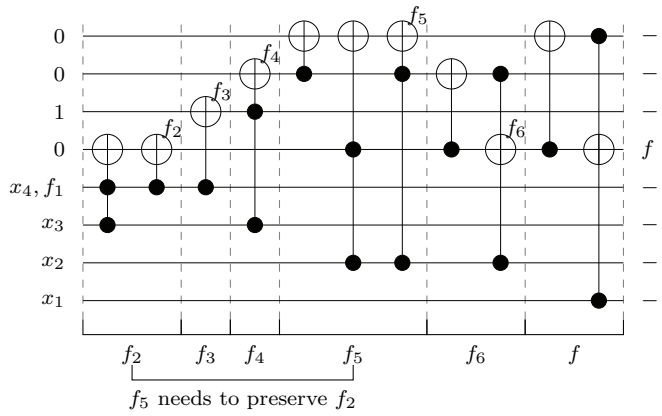
1. Traverse the DD in a depth-first manner.
2. For each inner node $v \in V$, generate a cascade of reversible gates which computes the (sub-)function represented by v. Output values of the previously traversed child nodes of v are utilized for this purpose.
3. Cascade all generated sub-circuits which eventually leads to a circuit realizing f.

The sub-circuits generated in Step 2 vary depending on decomposition type, types of child nodes, use of complement edges, etc. Fig. 3.1 provides a selection of cases that may occur in a DD together with their corresponding circuit patterns.

Example 3 *Fig. 3.2a shows a DD representing the function* $f = \overline{x}_1\overline{x}_2\overline{x}_3x_4 + \overline{x}_1x_2x_3\overline{x}_4 + x_1\overline{x}_2x_3\overline{x}_4 + x_1x_2\overline{x}_3x_4$ *as well as the respective co-factors resulting from the application*



(a) BDD



(b) Resulting Circuit

Figure 3.2.: Illustration of BDD-based Synthesis

of the Shannon decomposition. The co-factor f_1 can easily be represented by the primary input x_4. Having the value of f_1 available, the co-factor f_2 can be realized by the first two gates depicted in Fig. 3.2b[1]. In this fashion, respective sub-circuits can be added for all remaining co-factors until a circuit representing the overall function f results. The remaining steps are shown in Fig. 3.2b.

Thus far, only positive control lines have been considered in the DD-based synthesis. But as shown in previous work such as [WSPD12, DSR13, DRW$^+$13, ST13], additionally utilizing negative control lines may significantly reduce the number of gates as well as the resulting quantum costs of a reversible circuit. However, the utilization of negative control lines during DD-based synthesis has not been investigated yet. Because of this, significant potential for the improvement of DD-based synthesis has not been exploited.

[1]Note that an additional circuit line is added to preserve the values of x_4 and x_3 which are still needed by the co-factors f_3 and f_4, respectively.

In particular, the realization of complement edges or negative Davio decomposition may significantly profit from negative control lines.

In this work, these missing investigations and evaluations are performed. To this end, two complementary schemes are considered. First, it is evaluated whether corresponding post-synthesis approaches presented in the past can be applied in order to improve circuits obtained by DD-based synthesis. Second, negative control lines are explicitly considered during synthesis, i.e. an extended DD-based synthesis approach is proposed which directly applies negative control lines when mapping from nodes to sub-circuits. Experimental evaluations summarized in Section 3.4 confirm that both schemes lead to significant improvements.

3.2. Explicit Consideration of Negative Control Lines During Synthesis

In DD-based synthesis, negative control lines can explicitly be exploited for two purposes:

- Negative Davio decomposition can be realized in a similar fashion as positive Davio decomposition. They only differ in the polarity of the respective x_i-variable which, thanks to a negative control line, can easily be considered. This may lead to improvements since, as shown in Fig. 3.1, positive Davio can usually be realized with less gates and/or costs than negative Davio.
- Complemented edges can inherently be realized by negative control lines. In fact, complement edges are applied when the value of a sub-function to be considered shall be applied inversely. Again, this can easily be realized by the simple application of a negative control line, while, thus far, often additional logic has been required.

These observations are also confirmed by the realizations of the respective sub-circuits. More precisely, Fig. 3.3 shows the circuit realizations for all the cases previously discussed in Fig. 3.1 which additionally make use of negative control lines (the respective circuits have been obtained by the exact approach from [WSPD12] and represent minimal realizations with respect to the number of gates). Note that not all cases which might occur in DDs are enlisted in a pictorial fashion. Nevertheless, Table 3.1 lists the number of gates and the quantum costs for *all* possible cases[2] and, by this, allows a comprehensive comparison. Columns denoted by d provide the number of gates, while columns denoted by QC provide the respective quantum costs. Both columns are additionally distinguished between values obtained if positive control lines are considered only (pc) and if negative control lines are considered additionally (mc). The last rows (*Total*) list the sum of gates and quantum costs that could be saved considering negative control lines.

As can be seen, most of the patterns could be improved with respect to gate count and quantum costs. Especially the cases with complement edges (Table 3.1b) unveil

[2]Some node patterns are redundant and therefore not listed.

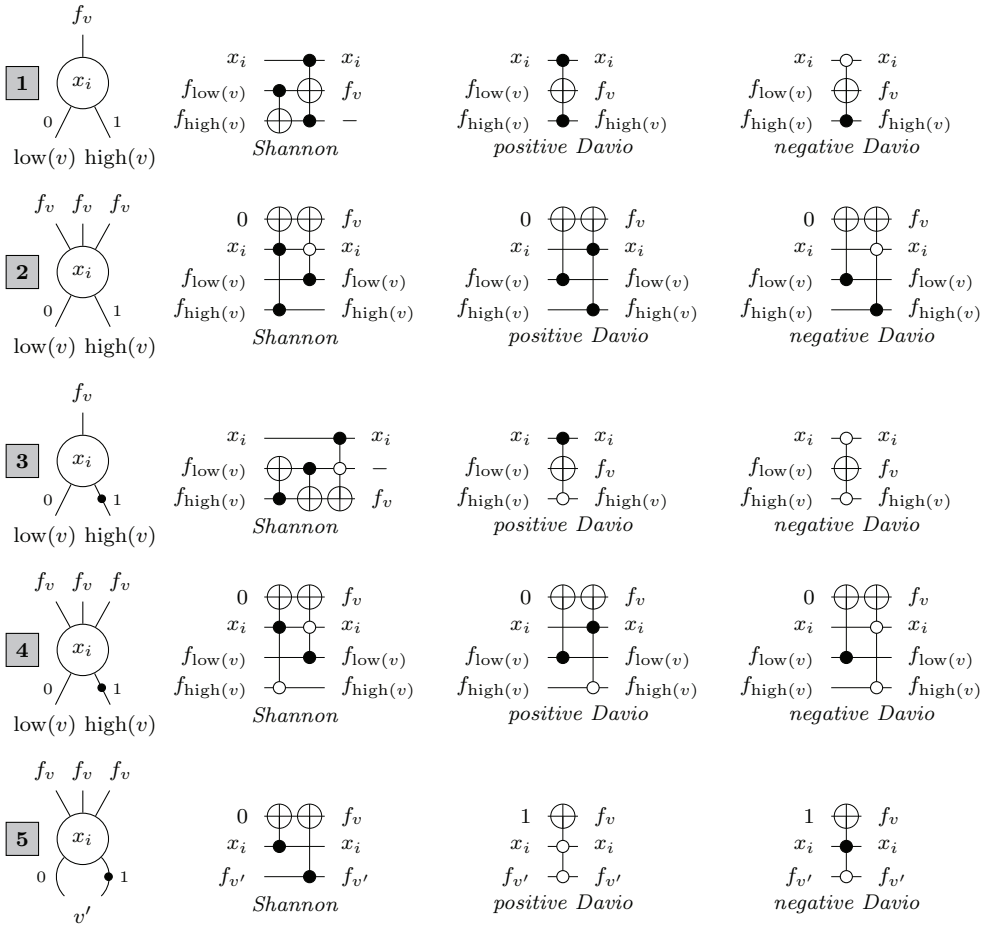


Figure 3.3.: Reversible Cascades (with Negative Control Lines) Representing the Different DD Decompositions

significant improvements. Interestingly, even some smaller realizations for the Shannon decomposition can be determined. In most of the cases, one gate – sometimes even two gates – can be saved. Quantum costs are improved by up to 4 in the best case. Considering that relatively small sub-circuits are considered which, however, are repeatedly applied during DD-based synthesis, this constitutes a significant improvement.

3.3. Post-Synthesis Optimization

Synthesis and optimization of reversible logic circuits have gained lots of attention in the past. As circuits generated from certain synthesis approaches require a large number of gates, there is a huge scope for post-synthesis optimization. In the literature, most of the optimization techniques relied on a gate library composed of positive control Toffoli gates only.

Table 3.1.: Gate Count and Quantum Cost for all DD Decompositions

(a) Without Complement Edges

	d		QC	
Case	pc	mc	pc	mc
LL_pD	2	**1**	6	**5**
LL_nD	1	1	5	5
LH_S_s	3	**2**	11	**10**
LH_pD_s	2	2	6	6
LH_nD_s	3	**2**	7	**6**
LH_S	2	2	6	6
LH_pD	1	1	5	5
LH_nD	2	**1**	6	**5**
1H_S	2	**1**	6	**5**
1H_pD	1	1	5	5
1H_nD	2	**1**	6	**5**
0H_S	1	1	5	5
L1_S	3	**1**	7	**6**
L1_pD	2	2	2	2
L1_nD	2	2	2	2
L0_S	2	**1**	6	**5**
11_pD	1	1	1	1
10_S	1	1	1	1
Total		9		8

(b) With Complement Edges

	d		QC	
Case	pc	mc	pc	mc
L-L_S_s	2	2	2	2
L-L_S	1	1	1	1
-LL_S_s	2	2	2	2
-LL_S	2	**1**	2	2
L-L_pD	3	**1**	7	**6**
L-L_nD	2	**1**	6	**5**
L-H_S_s	4	**2**	12	**10**
-LH_S_s	4	**2**	12	**10**
L-H_pD_s	3	**2**	7	**6**
L-H_nD_s	4	**2**	8	**7**
L-H_S	3	3	11	**7**
-LH_S	3	3	11	**8**
L-H_pD	2	**1**	6	**5**
L-H_nD	3	**1**	7	**6**
1-H_S	1	1	5	5
1-H_pD	2	**1**	6	**5**
1-H_nD	3	**1**	7	**6**
-L1_S	2	**1**	6	**5**
-L0_S	3	**1**	7	**6**
Total		20		21

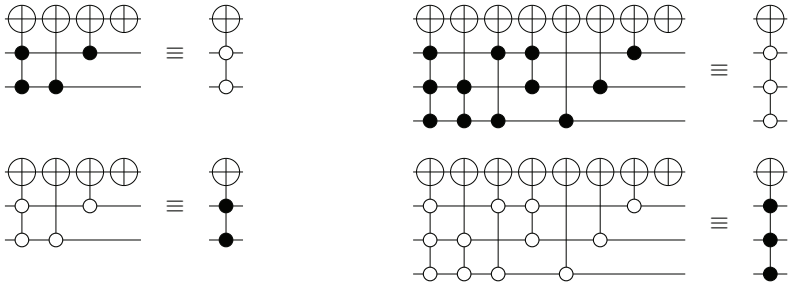


Figure 3.4.: Optimization Rules (Taken from [DRW$^+$13])

However, in a recent work [DRW$^+$13] it was shown that the power of negative control lines in Toffoli gates can be used quite elegantly to frame a set of template matching rules. Then, these rules can be applied in order to optimize a given reversible circuit. Some of these rules that may be used are illustrated in Fig. 3.4. Detailed experimental evaluations on circuits derived from various synthesis approaches demonstrated that significant reductions in the gate count and the quantum costs are possible when applying these rules.

Due to the nature of these rules, respective optimization methods usually perform better for circuits which inherit a specific structure, e.g. a clear separation between input lines and output lines. In previous work, this has successfully been shown on circuits generated by ESOP-based synthesis approaches (see [FTR07] for a general description of ESOP-based synthesis and [DRW$^+$13] for an evaluation on the corresponding post-synthesis optimization). However, an evaluation on circuits obtained by DD-based synthesis has not explicitly been conducted yet. Since also DD-based circuits inherit a rather regular structure, similar improvements are very likely. This is evaluated in detail later in Section 3.4.

3.4. Experimental Evaluation

The concepts and approaches discussed above have been evaluated. For this purpose, the post-synthesis optimization scheme proposed in [DRW$^+$13] has been applied. Additionally, the *RevKit*-implementations (taken from [SFWD12]) of the BDD-based and the KFDD-based synthesis approaches [WD09, SWD10] have been extended by the new cascades which are partially sketched in Fig. 3.3. All these approaches have eventually been evaluated using a set of benchmark functions taken from *RevLib* [WGT$^+$08].

Table 3.3a and Table 3.3b summarize the results for BDD-based synthesis and KFDD-based synthesis, respectively. The first two columns denote the name of the considered function as well as the number n of circuit lines generated by the respective approaches. Then, the remaining columns provide the number of gates (denoted by d) as well as the quantum costs (denoted by QC) of the circuits obtained by the original approach (i.e. the original BDD-based or KFDD-based synthesis) as well as the circuits obtained by applying the post-synthesis scheme (as discussed in Section 3.3) and the extended

Table 3.2.: Experimental Results for DD-based Synthesis Optimization

(a) BDD-based Synthesis Optimization

		Original [WD09]		Post-Synth. (Sec. 3.3)		Impr.		Explicit (Sec. 3.2)		Impr.	
Benchmark	n	d	QC	d	QC	d	QC	d	QC	d	QC
alu2_96	105	452	1436	358	1346	21%	6%	323	1233	29%	14%
alu4_98	541	2186	7222	1746	6784	20%	6%	1554	6476	29%	10%
apex2_101	498	1746	5922	1358	5534	22%	7%	1238	5462	29%	8%
apex5_104	1025	2909	10349	2246	9686	23%	6%	2059	9461	29%	9%
ex5p_154	206	647	1843	462	1659	29%	10%	372	1612	43%	13%
frg2_161	1219	3724	12468	2753	11497	26%	8%	2611	11404	30%	9%
hwb8_64	112	449	1461	346	1360	23%	7%	319	1289	29%	12%
hwb9_65	170	699	2275	540	2117	23%	7%	488	2001	30%	12%
seq_201	1617	5990	19362	4561	17935	24%	7%	3950	17390	34%	10%
spla_202	489	1709	5925	1321	5537	23%	7%	1217	5420	29%	9%
urf1_72	374	1848	6080	1441	5673	22%	7%	1354	5199	27%	14%
urf2_73	209	983	3187	764	2968	22%	7%	703	2720	28%	15%
urf3_75	668	3413	11357	2674	10618	22%	7%	2533	9743	26%	14%
urf5_76	216	860	2796	679	2616	21%	6%	607	2432	29%	13%
Average						23%	7%			30%	11%

(b) KFDD-based Synthesis Optimization

		Original [SWD10]		Post-Synth. (Sec. 3.3)		Impr.		Explicit (Sec. 3.2)		Impr.	
Benchmark	n	d	QC	d	QC	d	QC	d	QC	d	QC
alu2_96	107	326	894	257	836	21%	6%	212	806	35%	10%
alu4_98	452	1252	5216	1239	5206	1%	1%	1238	5243	1%	0%
apex2_101	394	949	3621	870	3555	8%	2%	823	3661	13%	-1%
apex5_104	1029	2088	9092	2011	9019	4%	1%	1984	9023	5%	1%
ex5p_154	202	419	1503	374	1459	11%	3%	367	1525	12%	-1%
frg2_161	1252	3311	9023	2587	8361	22%	7%	1920	8308	42%	8%
hwb8_64	115	337	1297	310	1270	8%	2%	334	1300	1%	0%
hwb9_65	170	513	1993	472	1952	8%	2%	510	1996	1%	0%
seq_201	828	2041	6469	1699	6200	17%	4%	1403	6214	31%	4%
spla_202	458	1116	3760	984	3644	12%	3%	854	3728	23%	1%
urf1_72	379	1614	4202	1278	3866	21%	8%	1312	3954	19%	6%
urf2_73	203	736	2420	581	2266	21%	6%	571	2290	22%	5%
urf3_75	665	2625	9149	2307	8831	12%	3%	2465	9048	6%	1%
urf5_76	207	700	1876	508	1687	27%	10%	457	1679	35%	11%
Average						14%	4%			18%	3%

approach (presented in Section 3.2). The columns *Impr.* provide the improvements with respect to the original realizations. All results have been generated in neglicable runtime, i.e. just a fraction of a second in most of the cases; the post-synthesis optimization scheme sometimes required slightly more time, but never more than 10 CPU seconds.

The results confirm the discussions from Section 3.1: The utilization of negative control lines significantly reduces the number of gates as well as the resulting quantum costs and, hence, indeed improves DD-based synthesis. In the best cases, up to 43% of the gates and 15% of the quantum costs can be saved. The improvements of circuits obtained by KFDD-based synthesis are somewhat slight. This can be explained by the fact that KFDD decomposition already leads to smaller circuits. Nevertheless, relevant improvements can also be observed here. Considering that these improvements come with no drawbacks, the application of negative control lines is a worthwhile addition to DD-based synthesis schemes.

3.5. Conclusion

In this chapter, we investigated the potential of utilizing negative control lines for DD-based synthesis. To this end, a post-synthesis scheme as well as an explicit consideration during synthesis have been inspected and evaluated. Experiments confirmed the expected improvements: Negative control lines indeed allow for the realization of reversible circuits with significantly less gate count and quantum costs. In the best cases, up to 43% of the gates and 15% of the quantum costs can be saved.

4. Mapping Conventional Circuits

This part of the thesis considers the design of reversible circuits under the full exploitation of the powerful design methods which exist for conventional circuits. At the beginning, the design of reversible circuits follows the design flow for conventional circuits. Afterwards, approaches to be developed will be applied which map the resulting conventional netlist to a reversible circuit description. In particular this mapping of a conventional circuit to a reversible circuit poses a serious challenge. Possible schemes for a mapping at the gate level and a mapping at the register transfer level are outlined next. Subsequently, the advantages and disadvantages of such a flow are discussed.

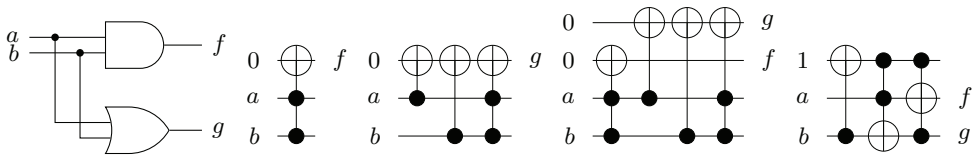
4.1. Mapping at the Gate Level

Mapping at the gate level is illustrated by the simple example in Fig. 4.1a showing a low level circuit representation in conventional logic. A simple mapping scheme could follow the procedure to substitute each conventional gate with their corresponding reversible counterpart[1]. Reversible realizations of the AND function and the OR function are provided in Fig. 4.1b and Fig. 4.1c, respectively. As they realize non-reversible functions, additional circuit lines (with a constant input 0) are neccessary. Simply composing these circuits leads to a functionally equivalent realization as shown in Fig. 4.1d.

4.2. Mapping at the Register Transfer Level

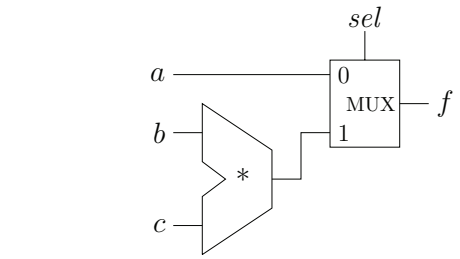
At the register transfer level, a circuit is described by a netlist of modules representing the data and control flow operations. Fig. 4.2(a) shows a simple example of a circuit in this abstraction level. In order to transform this circuit into a reversible equivalent, a mapping scheme similar to the one illustrated above for the gate level can be applied. The difference is just that complete modules rather than single gates have to be substituted.

[1] A similar scheme has been presented before in [ZRK07].

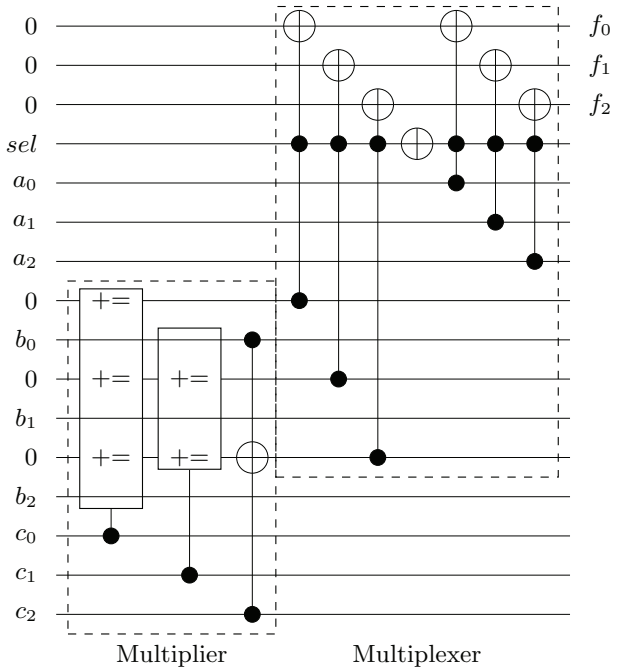


(a) Conv. Circuit (b) AND (c) OR (d) Composed (e) Minimal

Figure 4.1.: Mapping a Conv. Circuit to a Rev. Circuit at the Gate Level



(a) Conv. Circuit



(b) Composed Rev. Circuit

Figure 4.2.: Mapping a Conv. Circuit to a Rev. Circuit at the Register Transfer Level

For this purpose, past accomplishments in the design of reversible building blocks for various important data flow operations like adders, multipliers, etc. can be exploited (see e.g. [TG08]). For example, the multiplier depicted in Fig. 4.2a can be mapped to a reversible partial product realization illustrated at the left-hand side of Fig. 4.2b. The control flow, represented by modules like priority selectors or multiplexers, can similarly be realized as illustrated in Fig. 4.2 for the multiplexer module. Here, the value of the input labeled with 1 (0) is "copied" to the output signals iff the value of `sel` is 1 (0). These building blocks also require the availability of additional circuit lines with constant inputs (as can be seen in Fig. 4.2b)

4.3. Discussion

Following the scheme sketched above has the big advantage of allowing for an exploitation of the full power of conventional design methods which have been developed and in industrial use for several decades. But the resulting circuits suffer from the poor mapping methods that often just solely consider the respective gates or modules to be mapped. For example, the mapping sketched in Fig. 4.1 just solely maps two gates to corresponding cascades leading to the circuit depicted in Fig. 4.1d. But, in fact, a smaller circuit realizing the same functionality with fewer circuit lines and fewer gates can be found (depicted in Fig. 4.1e).

This drawback is less significant if the mapping is performed at the register transfer level. For the modules to be mapped here, dedicated designs are available. These save circuit lines and/or gates by considering the whole function at once instead of locally mapping single gates without acknowledging their relations to each other.

As the preliminary results summarized in Section 4.4 confirm, this scheme already leads to quite satisfactory results. But still, a significant amount of additional circuit lines with constant inputs is required.

Overall, exploiting the conventional design flow does not provide any support for reversibility until the resulting conventional circuit is mapped to its reversible equivalent. Therefore, the quality of the resulting circuit with respect to metrics relevant to reversible logic (like number of circuit lines or corresponding gate costs) almost entirely relies on the applied mapping and possibly applied post-synthesis optimization schemes. Improving these schemes is the major research challenge for this design direction.

4.4. Preliminary Evaluation

In order to evaluate the applicability of either design direction discussed above, preliminary implementations of the respective concepts have been created. More precisely, we implemented

- a basic mapping scheme which transforms a given conventional circuit at the register transfer level (synthesized from a Verilog description using RTLvision PRO 5.4.1 by Concept Engineering) to a corresponding reversible circuit (RTL TO REV. CIRCUIT)

as well as

- a basic synthesis scheme following the concepts proposed in [WOD10] and reviewed in Chapter 6 which generates a reversible circuit from a description in a reversible programming language (REV. CODE TO REV. CIRCUIT).

Results obtained by these implementations are provided in Table 4.1 for a selection of designs such as arithmetic logic units, a counter, circuits with a nested control structure, and others. Established cost metrics are considered for comparison, i.e. the number of lines (denoted by n), the number of gates (denoted by d), the quantum costs (QC), as well as the transistor costs (TC).

Table 4.1.: First Results for RTL to Rev. Circuit vs. Rev. Code to Rev. Circuit

	RTL to Rev. Circuit				Rev. Code to Rev. Circuit			
Benchmark	n	d	QC	TC	n	d	QC	TC
alu1_16	107	1079	7019	17776	117	1106	35463	39552
alu1_32	203	3935	27027	68208	229	3978	144791	154432
alu2_16	107	3632	147129	151376	117	3659	258872	234424
alu2_32	203	14416	1232073	1064464	229	14459	1704912	1402232
counter	57	106	494	1416	37	37	857	912
ite1_16	97	308	804	3424	34	210	1522	3816
ite1_32	193	628	1636	7008	66	434	3154	7912
ite2_16	194	680	1928	7872	37	422	6982	11000
ops1_16	128	1066	6122	16960	128	1066	6122	16960
ops1_32	256	3938	25282	66752	256	3938	25282	66752
ops2_16	128	764	6855	11824	112	633	1361	6512
ops2_32	256	1828	55007	56816	224	1305	2801	13424

These preliminary results unveil that, thus far, there is no clear indication whether scalable synthesis of reversible circuits should be conducted by the design flow discussed in Part I or the design flow discussed in Part II. For some designs (e.g. alu1_16), following the conventional design flow leads to better circuits. Other designs (e.g. ops2_32) benefit more from the reversible-specific design flow. Nevertheless, both are capable of *scalable* synthesis of reversible circuits. In fact, all circuits have been realized in negligable runtime (i.e. less than 1 CPU second). In contrast, previously proposed synthesis approaches (see e.g. [SM11]) are restricted by their Boolean data-structures in terms of truth tables or decision diagrams and, hence, are not scalable.

4.5. Conclusion

If a conventional circuit can be efficiently mapped to a reversible one, powerful methods from the conventional design flow can be utilized in the design of reversible circuits. We outlined a scheme for mapping at the gate level, which maps each gate individually and thus potentially creates a significant amount of additional lines and gates. To reduce this overhead, we then proposed a scheme for mapping at the register transfer level. This method already leads to adequate results similar to those of a basic synthesis scheme for reversible HDL. Its full potential, however, can only be learned by developing and optimizing the mapping.

From the preliminary evaluation, we get no clear lead whether the scalable design and synthesis of reversible circuits should follow the flow discussed in this chapter or the flow discussed in Part II. Nevertheless, we showed that both directions already allow for the design of large, complex reversible circuits.

Part II.

Developing a Specific Design Flow

5. The SyReC Language

In this chapter, the SyReC language is introduced. SyReC allows for the specification and the synthesis of complex logic through common HDL description means. Since every valid SyReC program is inherently reversible, the reversibility of the specification is ensured at the same time. The general concepts to achieve this are summarized in the first part of this chapter. Afterwards, the syntax and semantics of all SyReC description means are explained in detail.

5.1. General Concepts

In order to ensure reversibility in its description, SyReC adapts established concepts from the previously introduced reversible programming language Janus [YG07] and is additionally enhanced by hardware-related language constructs as it is targeting the description of reversible circuits. The general concepts of SyReC are summarized in the following.

5.1.1. Only Reversible Assignments

Being one of the most elementary language constructs, variable assignments such as used in the majority of the imperative languages are irreversible and can therefore not be part of a reversible language. The concept of *reversible assignments* (or sometimes also called reversible updates) is used as an alternative. Reversible assignments have the form $v \oplus = e$ with $\oplus \in \{\hat{}, +, -\}$ such that the variable v does not appear in the right-hand side expression e. Although SyReC is limited to this set of operators, in general any operator f can be used for the reversible assignment, if there exists an inverse operator f^{-1} such that

$$v = f^{-1}(f(v, e), e) \tag{5.1}$$

for all variables v and for all expressions e. Note that '+' (addition) is inverse to '-' (subtraction), and vice versa, and '^' (bitwise exclusive OR) is inverse to itself. When executing the program in reverse order, all reversible assignment operators are replaced by their inverse operators.

5.1.2. Syntactical Expressiveness

Due to the construction of the reversible assignment, the right-hand side expression can also be irreversible and compute any operation. The most common operations are directly applicable using a wide variety of syntax including arithmetic (+, *, /, %, *>),

bitwise (&, |, ^), logical (&&, ||), and relational (<, >, =, !=, <=, >=) operations. The reversibility is ensured, since the input values to the operation are also given to the inverse operation when reverting the assignment (cf. (5.1)). In order to specify e.g. a multiplication a*b, a new free signal c must be introduced which is used to store the result (i.e. c^=(a*b) is applied).

5.1.3. Reversible Control Flow

A reversible data flow is ensured due to the above mentioned assignment operations, and the control flow is made bijectively executable in a similar fashion. This becomes particularly manifest in conditional statements. In contrast to non-reversible languages, SyReC requires an additional *fi*-condition for each *if*-condition which is applied as an assertion. This *fi*-condition is required, since a conditional statement may not be computed in both directions using the same condition, i.e. it cannot be ensured that the same block (*then*-block or *else*-block) is processed when computing an *if*-statement in the reverse direction. As a solution, a *fi*-condition that is asserted when computing the statement in the reverse direction is added ensuring a consistent execution semantic. This language principle is illustrated in more detail in the next section.

5.1.4. Specific Hardware Description Properties

Since SyReC is used for the synthesis of reversible circuits, it obeys some HDL related properties:

- The single data-type is a circuit signal with parameterized bit width.
- Access to single bits ($x.N$), a range of bits ($x.N$:N), as well as the size (#x) of a signal is provided.
- Since loops must be completely unrolled during synthesis, the number of iterations has to be available before compilation. That is, dynamic loops (defined by expressions) are not allowed.
- Further operations as used in hardware design (e.g. shifts '<<' and '>>') are provided.

Overall, the implementation of all these general concepts led to the SyReC syntax as defined by means of the EBNF in Fig. 5.1. In the following, the syntax and the semantics of all description means are explained and illustrated in detail.

Program and Modules

1 ⟨*program*⟩ ::= ⟨*module*⟩ {⟨*module*⟩}

2 ⟨*module*⟩ ::= **'module'** ⟨*identifier*⟩ **'('** [⟨*parameter-list*⟩] **')'** {⟨*signal-list*⟩} ⟨*statement-list*⟩

3 ⟨*parameter-list*⟩ ::= ⟨*parameter*⟩ {**','** ⟨*parameter*⟩}

4 ⟨*parameter*⟩ ::= (**'in'** | **'out'** | **'inout'**) ⟨*signal-declaration*⟩

5 ⟨*signal-list*⟩ ::= (**'wire'** | **'state'**) ⟨*signal-declaration*⟩ {**','** ⟨*signal-declaration*⟩}

6 ⟨*signal-declaration*⟩ ::= ⟨*identifier*⟩ {**'['**⟨*int*⟩**']'**} [**'('**⟨*int*⟩**')'**]

Statements

7 ⟨*statement-list*⟩ ::= ⟨*statement*⟩ {**';'** ⟨*statement*⟩}

8 ⟨*statement*⟩ ::= ⟨*call-statement*⟩ | ⟨*for-statement*⟩ | ⟨*if-statement*⟩ | ⟨*unary-statement*⟩ | ⟨*assign-statement*⟩ | ⟨*swap-statement*⟩ | ⟨*skip-statement*⟩

9 ⟨*call-statement*⟩ ::= (**'call'** | **'uncall'**) ⟨*identifier*⟩ **'('** (⟨*identifier*⟩ {**','** ⟨*identifier*⟩}) **')'**

10 ⟨*for-statement*⟩ ::= **'for'** [[**'$'** ⟨*identifier*⟩ **'='**] ⟨*number*⟩ **'to'**] ⟨*number*⟩ [**'step'** [**'-'**] ⟨*number*⟩] ⟨*statement-list*⟩ **'rof'**

11 ⟨*if-statement*⟩ ::= **'if'** ⟨*expression*⟩ **'then'** ⟨*statement-list*⟩ **'else'** ⟨*statement-list*⟩ **'fi'** ⟨*expression*⟩

12 ⟨*assign-statement*⟩ ::= ⟨*signal*⟩ (**'^'** | **'+'** | **'-'**) **'='** ⟨*expression*⟩

13 ⟨*unary-statement*⟩ ::= (**'~'** | **'++'** | **'--'**) **'='** ⟨*signal*⟩

14 ⟨*swap-statement*⟩ ::= ⟨*signal*⟩ **'<=>'** ⟨*signal*⟩

15 ⟨*skip-statement*⟩ ::= **'skip'**

16 ⟨*signal*⟩ ::= ⟨*identifier*⟩ {**'['** ⟨*expression*⟩ **']'**} [**'.'** ⟨*number*⟩ [**':'** ⟨*number*⟩]]

Expressions

17 ⟨*expression*⟩ ::= ⟨*number*⟩ | ⟨*signal*⟩ | ⟨*binary-expression*⟩ | ⟨*unary-expression*⟩ | ⟨*shift-expression*⟩

18 ⟨*binary-expression*⟩ ::= **'('** ⟨*expression*⟩ (**'+'** | **'-'** | **'^'** | **'*'** | **'/'** | **'%'** | **'*>'** | **'&&'** | **'||'** | **'&'** | **'|'** | **'<'** | **'>'** | **'='** | **'!='** | **'<='** | **'>='**) ⟨*expression*⟩ **')'**

19 ⟨*unary-expression*⟩ ::= (**'!'** | **'~'**) ⟨*expression*⟩

20 ⟨*shift-expression*⟩ ::= **'('** ⟨*expression*⟩ (**'<<'** | **'>>'**) ⟨*number*⟩ **')'**

Identifier and Constants

21 ⟨*letter*⟩ ::= (**'A'** | ... | **'Z'** | **'a'** | ... | **'z'**)

22 ⟨*digit*⟩ ::= (**'0'** | ... | **'9'**)

23 ⟨*identifier*⟩ ::= (**'_'** | ⟨*letter*⟩) {(**'_'** | ⟨*letter*⟩ | ⟨*digit*⟩)}

24 ⟨*int*⟩ ::= ⟨*digit*⟩ {⟨*digit*⟩}

25 ⟨*number*⟩ ::= ⟨*int*⟩ | **'#'** ⟨*identifier*⟩ | **'$'** ⟨*identifier*⟩ | (**'('** ⟨*number*⟩ (**'+'** | **'-'** | **'*'** | **'/'**) ⟨*number*⟩ **')'**)

Figure 5.1.: Syntax of the Hardware Description Language SyReC

5.2. Module and Signal Declarations

Each SyReC specification (denoted by ⟨*program*⟩ in Line 1 in Fig. 5.1) consists of one or more *modules* (denoted by ⟨*module*⟩ in Line 2). A module is introduced with the keyword *module* and includes an identifier (represented by a string as defined in Line 23), a list of parameters representing global signals (denoted by ⟨*parameter-list*⟩ in Line 3), local signal declarations (denoted by ⟨*signal-list*⟩ in Line 5), and a sequence of statements

Table 5.1.: SyReC's Signal Access Modifiers and Implied Circuit Properties

Modifier	Constant Input	Garbage Output	State	Initial Value
in	–	yes	no	given by primary input
out	0	no	no	0
inout	–	no	no	given by primary input
wire	0	yes	no	0
state	–	no	yes	given by pseudo-primary input

(denoted by ⟨*statement-list*⟩ in Line 7). The top-module of a program is defined by the special identifier *main*. If no module with this name exists, the last module declared is used as the top-module instead.

SyReC uses a *signal* representing a non-negative integer as its sole data type. The bit width of signals can optionally be defined by round brackets after the signal name (Line 6). If no bit width is specified, a default value is assumed. For each signal, an *access modifier* has to be defined. For a parameter signal (used in a module declaration), this can be either *in*, *out*, or *inout* (Line 4). Local signals can either work as internal signals (denoted by *wire*) or in case of sequential circuits as state signals[1] (denoted by *state*; Line 5). The access modifier affects properties in the synthesized circuits as summarized in Table 5.1. Besides that, signals can be grouped into multi-dimensional arrays of constant length using square brackets after the signal name and before the optional bit width declaration (Line 6).

Example 4 *Fig. 5.2 shows several module declarations possible in SyReC including an adder-module with two inputs and one output (adder1), an adder-module with fixed bit widths for the inputs and outputs (adder2), an adder-module where four operands are given by a 4-segment array composed of 16-bit signals (adder3), and an arbitrary module with local and state signals (myCircuit).*

```
module adder1(in a, in b, out c)

module adder2(in a(16), in b(16), out c(16))

module adder3(in inputs[4](16), out c(16))

module myCircuit(in input1, in input2, out output)
  wire auxSignal(16)
  state stateSignal
```

Figure 5.2.: Module Declarations in SyReC

[1] Note that, depending on the application, feedback and, hence, state signals might not be allowed in reversible circuits. Nevertheless, SyReC supports this concept in principle. For a more detailed discussion on reversible sequential circuits, we refer to [CW07, LP09].

```
wire a, b, c
call adder1(a, b, c)
```

Figure 5.3.: Calling a Module Identified by adder1 in SyReC

5.3. Statements

Statements include call and uncall of other modules, loops, conditional statements, and various data operations (i.e. reversible assignment operations, unary operations, and swap statements; Line 8). The empty statement can explicitly be modeled using the *skip* keyword (Line 15). Statements are separated by semicolons (Line 7). Signals within statements are denoted by ⟨*signal*⟩ allowing access to the whole signal (e.g. `x`), a certain bit (e.g. `x.4`), or a range of bits (e.g. `x.2:4`, Line 16). The bit width of a signal can also be accessed (e.g. `#x`; Line 25).

5.3.1. Call and Uncall of Modules

Hierarchic descriptions are realized in SyReC by means of modules which can be called and uncalled. For this purpose, the keyword *call* (*uncall*) has to be applied together with the identifier of the module to be called and its parameters (Line 9). Call executes the selected module in forward direction, while uncall executes the selected module backwards.

Example 5 *If a SyReC description of an adder is available (as e.g. declared in Fig. 5.2), it can be added to a design by the call command as shown in Fig. 5.3.*

5.3.2. Loops

An iterative execution of a block is defined by means of loops (defined in Line 10). The number of iterations has to be available prior to the compilation, i.e. dynamic loops are not allowed. Therefore, e.g. fix integer values, the bit width of a signal, or internal (local) *$*-variables can be applied. Furthermore, the current value of internal counter variables can be accessed during the iterations. Using the optional keyword *step*, also the iteration itself can be modified. A loop is terminated by *rof*.

Example 6 *Fig. 5.4 shows several loop descriptions possible in SyReC including (a) a simple loop with 10 iterations, (b) an iteration over all bits of an n-bit signal, and (c) a loop with a step definition.*

```
for 1 to 10 do
  // statements
rof
```

(a) Simple Loop

```
wire x
for $i = 0 to #x do
  // statements (possibly using $i)
  // the ith bit of x can be accessed by x.$i
  // a range of bits can be accessed e.g. by x.0:$i
rof
```

(b) Loop over Bits of a Signal

```
for $counter = 1 to 10 step 2 do
  // statements
  // the loops iterates 5 times
  // (i.e., $counter is set to 1, 3, 5, 7, and 9 only)
rof
```

(c) Loop with *step* Keyword

Figure 5.4.: Loops in SyReC

5.3.3. Conditional Statements

Conditional statements (defined in Line 11) need an expression to be evaluated followed by the respective *then*- and *else*-block. Each of these blocks is a sequence of statements. In a forward computation, the *then*-block is executed if, and only if, the *if*-expression evaluates to 1; otherwise, the *else*-block is executed. In order to ensure reversibility, a conditional statement is terminated by a *fi* together with an adjusted expression. In a backward computation, the *fi*-expressions decides whether the *then*- or the *else*-block is reversibly executed. In case neither the *then*- nor the *else*-bock modifies an input value of the conditional expression, the *if*- and the *fi*-expression are identical.

Example 7 *Fig. 5.5 shows two different conditional statements in SyReC. The first one does not modify any of the inputs of the conditional expressions (signal b in this case). Hence, the* if- *and the* fi-*expression are identical. In contrast, the* then-*block of the second conditional statement modifies the value of signal b. Hence, a suitable* fi-*expression different from the* if-*expression has to be provided to ensure correct execution semantics in both directions.*

```
if (b = 5) then
  x += y // executed if b = 5
else
  x -= y // executed if b != 5
fi (b = 5);

if (b = 5) then
  b += y // executed if b = 5 (fwd) or b = 5 + y (bwd)
else
  x -= y // executed otherwise
fi (b = (5 + y))
```

Figure 5.5.: Conditional Statements in SyReC

Table 5.2.: Semantics of Assignment Statements in SyReC

Operation	Semantic
x `^=` e	Bitwise XOR assignment of e to x, i.e. $x := x \oplus e$
x `+=` e	Increase by value of e to x, i.e. $x := x + e$
x `-=` e	Decrease by value of e to x, i.e. $x := x - e$
`~=` x	Bitwise inversion of x
`++=` x	Increment of x
`--=` x	Decrement of x
x `<=>` y	Swapping value of x with value of y

5.3.4. Assignment Statements

All further statements include the reversible assignment statements (denoted by ⟨*assign-statement*⟩), unary statements (denoted by ⟨*unary-statement*⟩), and the swap statement (denoted by ⟨*swap-statement*⟩) as defined in Line 12 to Line 14. The semantics of these statements is summarized in Table 5.2, whereby signals are denoted by x, y and expressions are denoted by e. Since these statements perform only reversible operations, they may assign new values to signals. Therefore, the respective signal(s) to be modified must not appear in the expression on the right-hand side.

Example 8 *Fig. 5.6 shows some of these statements in action. It can easily be seen that all these operations can be executed in both directions, i.e. forward and backward computation always lead to unique results.*

```
b += 5;  // b := b+5
a ^= b;  // a := a^b
~= a;    // a := bitwise inversion of a
++= c;   // c := c+1
a <=> c  // a := c and c := a
```

Figure 5.6.: Assignment, Unary, and Swap Statements in SyReC

Table 5.3.: Semantics of Expressions in SyReC

Operation	Semantic
e + f	Addition of e and f
e - f	Subtraction of e and f
e * f	Lower bits of multiplication of e and f
e *> f	Upper bits of multiplication of e and f
e / f	Division of e and f
e % f	Remainder of division of e and f
e ^ f	Bitwise XOR of e and f
e & f	Bitwise AND of e and f
e | f	Bitwise OR of e and f
~e	Bitwise inversion of e
e && f	Logical AND of e and f
e || f	Logical OR of e and f
!e	Logical NOT of e
e < f	True, if, and only if, e is less than f
e > f	True, if, and only if, e is greater than f
e = f	True, if, and only if, e equals f
e != f	True, if, and only if, e not equals f
e <= f	True, if, and only if, e is less or equal to f
e >= f	True, if, and only if, e is greater or equal to f
e << N	Logical left shift of e by N
e >> N	Logical right shift of e by N

5.4. Expressions

Expressions as defined in Line 17 to Line 20 are applied e.g. in the right-hand side of assignment statements or as branching condition in *if*/*fi*-statements. Since expressions do not modify the values of any signal, also non-reversible operations can be applied in expressions without jeopardizing the reversibility. By this, a wide range of different description means is provided. Table 5.3 lists the semantic of all operations which can be used in expressions, whereby sub-expressions are denoted by e, f and natural numbers are denoted by N.

Example 9 *Fig. 5.7 shows some statements including expressions that demonstrate the range of description means available in SyReC. Although the language is restricted in order to ensure reversibility (e.g. statements such as c=a*b are not allowed), common functionality can easily be specified nevertheless (e.g. with a new free signal c and c^=a*b). It can easily be seen that, despite of the usage of non-reversible operations in Fig. 5.7, all statements still can be executed in both directions.*

```
c ^= (a * b); // c := a*b if c is a new free signal
x.0 ^= ((a > 3) && (b != 0));
x.1:3 ^= (c.0:2 | 4);
if ((a + b) <= 10) then
  c += (3 * b)
else
  c -= (a % 2)
fi ((a + b) <= 10)
```

Figure 5.7.: Application of Expressions in SyReC

5.5. Conclusion

Using the language introduced in this chapter, it is possible to specify reversible circuits on a higher level of abstraction. In particular for the design of complex functionality, SyReC clearly outperforms currently applied description means such as truth tables, permutations, and decision diagrams. Later in Chapter 9 this is further demonstrated by means of a complete design of a processor in SyReC. Beforehand, the synthesis of a reversible circuit based on a SyReC description is introduced.

6. SyReC Synthesis

Given a SyReC specification, a crucial question is how to obtain a reversible circuit realization. We use a hierarchical synthesis approach to handle the expressive power of the language. In this chapter, the synthesis of a SyReC program is explained. To give all the details, Chapter 7 provides the realization of each language element that is not discussed here. Due to the hierarchical approach, an overhead regarding gates and circuit lines is generated in most circuit realizations. Chapter 8 describes methods and possibilities for optimizing the synthesis result.

This chapter is structured as follows. After introducing the general concept, details on the synthesis process are given. Section 6.2 describes the realization of assignment statements, including unary statements and swap statements. The synthesis of expressions, which occur in assign statements as well as in conditional statements, is covered by Section 6.3. Finally, Section 6.4 describes the realization of the control logic, i.e. call/uncall statements, loops, and conditional statements.

6.1. General Concept

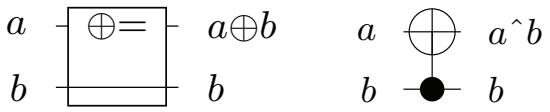
In order to synthesize a given SyReC specification, we developed a hierarchical synthesis method. Our approach traverses the whole program and adds a sub-circuit for each statement realizing the respective functionality. These sub-circuits are composed of existing realizations, so-called *building blocks*, of the individual statements and operations used in expressions. By adjusting the building blocks to the signals' bit width and applying them to the corresponding circuit lines, the desired realization is generated.

If a SyReC program consists of various modules, a main module is defined as the starting point of the synthesis. Other modules are synthesized when they are called or uncalled by the respective statements. All signals are realized by buses of joint reversible circuit lines with the specified bit width.

In the following, the individual mappings of the statements to the respective reversible cascades are described. We distinguish between the synthesis of (A) assignment statements (including unary statements and swap statements), (B) expressions, as well as (C) control logic including call/uncall, loops, and conditional statements.

6.2. Synthesis of Assignment Statements

To change the value of a signal, assignment statements are applied. These include assignments of the form $x \oplus= e$ with $\oplus \in \{\texttt{\^{}}, \texttt{+}, \texttt{-}\}$, unary statements of the form $\otimes= x$ with



(a) General: $a \oplus = b$ (b) $a \hat{} = b$

Figure 6.1.: Synthesis of Assignment Statements

$\otimes \in \{$`~`,`++`,`--`$\}$, and swap statements of the form x `<=>` y. Since each SyReC statement must be reversible, signal values are not overwritten but rather updated with a new value. The old value can be recovered by applying the inverted assignment operation.

6.2.1. Swap Statements

A swap statement `x <=> y` is applied to exchange the values of two signals x and y. To synthesize this statement, the signals are evaluated first, i.e. the corresponding circuit lines for x and y are determined (see Section 6.2.4). Then, the signal values are exchanged bit by bit. This is done by simply adding a SWAP gate to each pair of lines (x_i, y_i) with $0 \leq i \leq n-1$.

6.2.2. Unary Statements

A unary statement $\otimes = x$ is used to invert, increment or decrement the value of a single signal. Again, the first step is to evaluate the signal x. To implement the operation $\otimes \in \{$`~`,`++`,`--`$\}$, a building block is applied to the lines corresponding to x. These building blocks are described in Section 7.1.

6.2.3. Reversible Assignments

A reversible assignment of the form $x \oplus = e$ is used to increase or decrease the value of x by the value of e, or to compute a bitwise XOR of the values of x and e. If the value of x is 0, the XOR assignment $x \hat{} = e$ is equivalent to a regular assignment $x = e$.

For the implementation of this kind of statement, x and e have to be evaluated. Unlike the signal x, the right hand side e is a general expression, i.e. it could be a term like $(a * b) + c$. In this case, a sub-circuit is generated to compute the resulting value of e, as explained in Section 6.3. Having evaluated both x and e, a building block is applied to the corresponding lines to implement the actual assignment.

In the following, we use the notation depicted in Fig. 6.1a to denote such an operation in a circuit structure. Solid lines that cross the box represent the signals(s) on the right hand side of the statement, i.e. the signal(s) whose values are preserved.

The simplest reversible assignment operation is the bitwise XOR (e.g. $a \hat{} = b$). For 1-bit signals, this operation can be synthesized by a single Toffoli gate as shown in Fig. 6.1b. If signals with a bit width greater than 1 are applied, for each bit a Toffoli gate is applied analogously. Details on the implementation of all assignment operations can be found in Section 7.2.

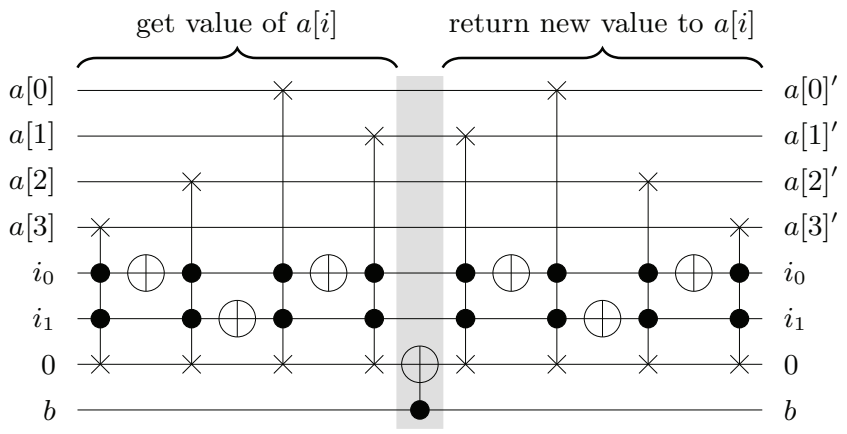


Figure 6.2.: Synthesizing `a[i] ^= b`

6.2.4. Evaluation of Signals

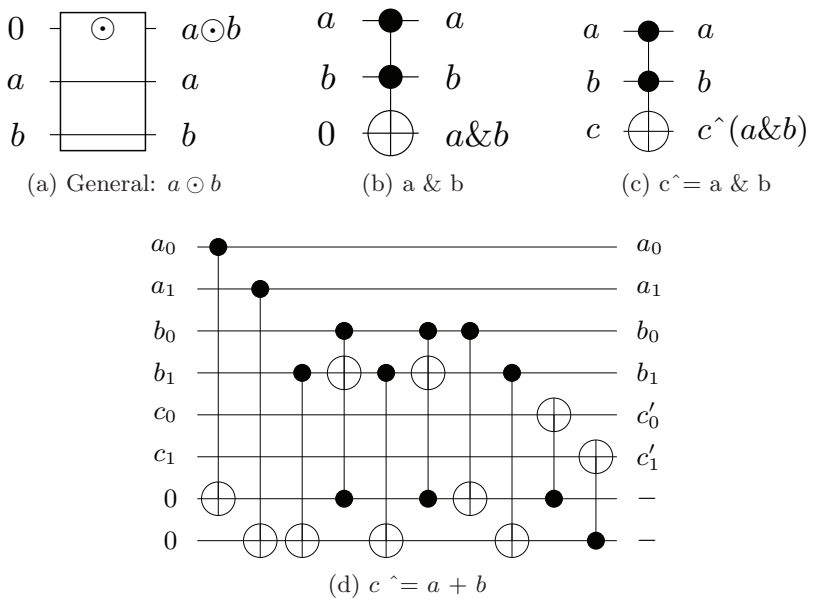
In most cases, the evaluation of a SyReC signal is trivial. If a whole signal (e.g. `x`), a single bit of a signal (e.g. `x.1`), or a range of bits of a signal (e.g. `x.3:6`) is accessed, the corresponding circuit lines are simply looked up in a map.

The only exception is the dynamical access to an array element, e.g. in the statement `a[i] ^= b`. Assuming i is a signal with a value depending on the primary inputs, the lines corresponding to `a[i]` can not be determined during synthesis. In this case, a sub-circuit as shown in Fig. 6.2 is applied. Here, the array a has four elements and the signal i consists of two bits. An additional line is used for the computation.

In the first part of the circuit, the value of the additional line is swapped with the value of `a[i]`. This is done by adding Fredkin gates which are controlled by the value of i. For example, if $i = 11$, only the first gate is activated, and the value of the additional line is swapped with the value of $a[3]$. If $i = 10$, only the second Fredkin gate is activated, exchanging the values of the additional line and $a[2]$, and so on. Note that the value of the additional line is depicted as 0, but can actually be arbitrary, as it has no influence on the computation.

In the middle part of the circuit, the line corresponding to `a[i]` is known to be the additional line (or lines, depending on the bit width of a). Now, the assignment operation can be executed. In this example, a single CNOT gate computes `a[i] ^= b`.

After the value of `a[i]` is updated according to the statement, it has to be returned to its original place (i.e. circuit line). Thus, in the last part of the circuit, the values of the additional line and `a[i]` are switched again. This is done by applying the inverse of the first part of the circuit. As a result, the value of `a[i]` is updated and the values of i and the additional line are restored.



(a) General: $a \odot b$ (b) a & b (c) c^= a & b

(d) $c \hat{}= a + b$

Figure 6.3.: Synthesis of Expressions

6.3. Synthesis of Expressions

Expressions include operations that are not necessarily reversible, like multiplication or bitwise conjunction. To denote such binary operations in a circuit structure, the notation depicted in Fig. 6.3a is used in the following. Again, solid lines represent the signals whose values are preserved. In this case, all input signals' values have to be kept to make the computation reversible and to avoid errors in case the signals are used in further statements.

Synthesis of irreversible functions in reversible logic is not a new issue, so reversible circuit realizations already exist for most operations considered here. To make an irreversible function reversible, additional lines with constant inputs are applied (see e.g. [MD04a, WKD11]). As an example, Fig. 6.3b shows a reversible gate that realizes an AND operation. As can be seen, this requires one additional circuit line with a constant input value 0. Similar mappings exist for all other operations (see Section 7.3).

Since expressions occur within statements, a more compact realization is possible in some cases. For example, the statement `c ^= (a & b)` can be realized by a single gate as shown in Fig. 6.3c. Compared to the standard approach of computing `a & b` first and then assigning it to c, this realization requires half the gates and no additional circuit line. However, such a simple combination is not possible for all statements. As an example, Fig. 6.3d shows a two-bit addition whose result is applied to a bitwise XOR, i.e. `c ^= (a + b)`. Here, removing the constant lines and directly applying the XOR operation on the lines representing c would lead to a wrong result. This is because intermediate results are stored on the lines representing the sum. Since these values are

used later, performing the XOR operation "in parallel" would destroy the result. Thus, to have a combined realization of a bitwise XOR and an addition, a precise embedding for this case must be generated. Since determining the embeddings and circuits for arbitrary combinations of statements and expressions is a cumbersome task, constant lines are applied to realize the respective functionality step by step.

6.4. Synthesis of the Control Logic

The control flow of a SyReC program can be defined by loops, call/uncall statements, and conditional statements. While the implementation of loops and calls is quite straightforward, conditional statements can be realized in various ways. We propose two complementary variants, one saving circuit lines and the other saving gate cost.

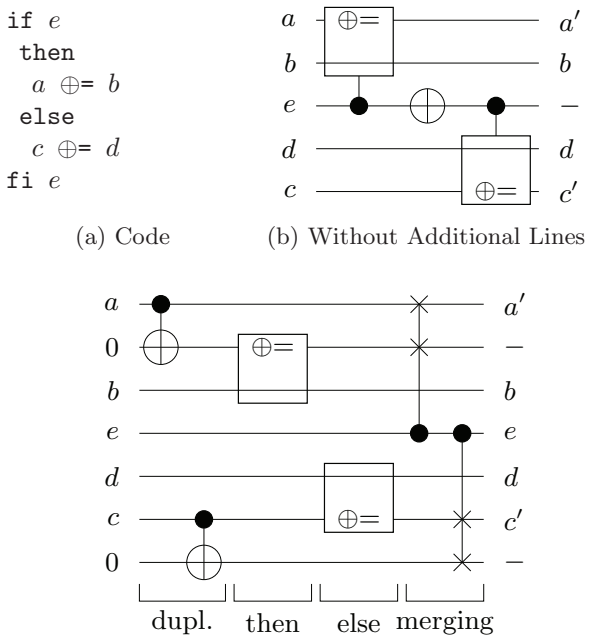
6.4.1. Loops and Calls

Loops are realized in a straightforward way, namely unrolling them. As elaborated in Section 5.3.2, the number of iterations of a loop is fixed and known prior to the synthesis. Thus for each iteration, the statements in the loop body are synthesized anew – if applicable, current values of loop variables are inserted.

Call and uncall of modules are handled similarly. If a module m is called, all statements in m are synthesized using the given signal parameters. Local signals in m are realized with additional lines. If a module m is uncalled, the desired function is to execute m backwards, or apply the inverse of m. To realize this, the statements in m are synthesized in reverse order, while each statement is reversed itself. For example, `c += (a * b)` would be converted to `c -= (a * b)` before synthesis.

6.4.2. Conditional Statements

To realize conditional statements (i.e. *if*-statements as introduced in Section 5.3.3), two complementary variants are proposed. The first one is depicted in Fig. 6.4b. Here, the statements in the *then*- and *else*-block are mapped to reversible cascades with an additional control line added to all gates. Thus, the respective operations of the statements in the *then*-block (*else*-block) are computed if and only if the result of the expression (stored in signal e) is 1 (0). A NOT gate is applied to flip the value of e so that the gates of the *else*-block can be "controlled" as well.



(a) Code (b) Without Additional Lines

(c) With Additional Lines

Figure 6.4.: Synthesis of Conditional Statements

Fig. 6.4c shows the second realization of a conditional statement, which is realized in three steps:

1. All signals in the *then*- or *else*-block, which potentially are assigned a new value (e.g. that are on the left-hand side of a reversible assignment operation), are duplicated. This requires an additional circuit line with constant input 0.

2. The statements within the blocks are mapped to reversible cascades. The duplications introduced in the previous step are applied to intermediately store the results of the *then*-block and the original values of the signals in the *else*-block.

3. Depending on the result of the conditional expression e, the values of the duplicated lines and the original lines are swapped. More precisely, in the example of Fig. 6.4a the value of a is swapped with its (newly assigned) duplication if e evaluates to 1. Analogously, if e evaluates to 0 the (newly assigned) value of c is passed through unaltered.

Having both realizations, it is up to the designer which one should be applied during synthesis. The second realization leads to additional circuit lines in contrast to the first realization. However, due to the additional control lines both the quantum cost and the transistor cost of the circuit significantly increase in the first realization. Besides other aspects, this is further evaluated in Section 8.3.

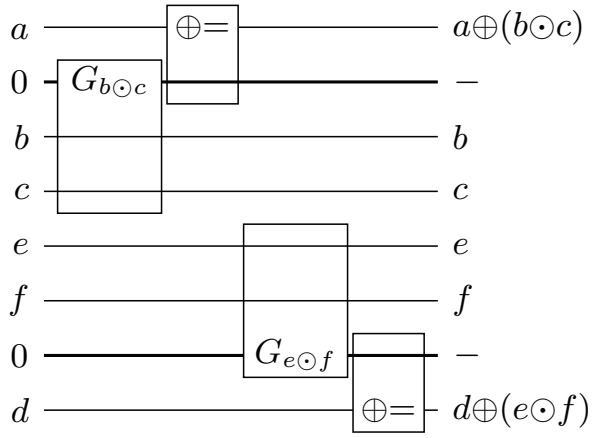


Figure 6.5.: Circuit Structure Generated by SyReC Synthesis

6.5. Conclusion

With the procedure explained in this chapter, it is possible to automatically synthesize reversible circuits specified in SyReC. In general, the following two steps are performed for each statement:

1. Compose a sub-circuit $G_{\odot}$ realizing all the expressions in a statement using the respective building blocks. The result of the expressions is buffered by means of additional circuit lines.

2. Compose a sub-circuit $G_{\oplus}$ realizing the overall statement using the existing building blocks of the statement itself together with the buffered results of the expressions.

Hence, the resulting circuits basically have a structure as shown in Fig. 6.5, i.e. cascades of building blocks for the respective assignment statements and their expressions results.

Obviously, this leads to a significant number of additional circuit lines with constant inputs which are used to buffer intermediate results of the expressions. The precise number of additional circuit lines increases with respect to the complexity of the expression. Usually, a large number of circuit lines is seen as a disadvantage. However, later in Section 8.1 an extended synthesis scheme is presented that removes many of these additional lines.

7. SyReC Building Blocks

This chapter provides a detailled description of the building blocks used in the synthesis of SyReC programs. For the circuit realization we chose to use the MCT library, i.e. multiple control Toffoli gates with positive controls only. Since the building blocks correspond to SyReC operations, they have to be automatically expandable to arbitrary bit widths of their input variables. Minimal solutions can be found for small bit widths, but practically not be generated for any bit width. Thus, some of the presented solutions come with a large overhead in terms of gate count and gate cost. However, this allows for the automatic synthesis of functions which can not be handled by most reversible circuit synthesis methods. Moreover, due to the hierarchical synthesis scheme, each building block can easily be replaced if a more efficient design is found.

First, the realizations of the unary statements are described in Sec. 7.1. Those include the bitwise negation as well as the increment and decrement of a variable. Sec. 7.2 illustrates the building blocks used for the reversible assignments, i.e. XOR assignment, increase, and decrease. Finally, the realizations for all unary and binary operations used within SyReC expressions are given in Sec. 7.3.

7.1. Unary Statements

There are three unary statements in the SyReC language, namely bitwise negation, increment, and decrement. As these statements change the value of a single input variable, with no other variables involved, they can be realized by building blocks operating directly on the variable's lines. Details on these building blocks are given in this section.

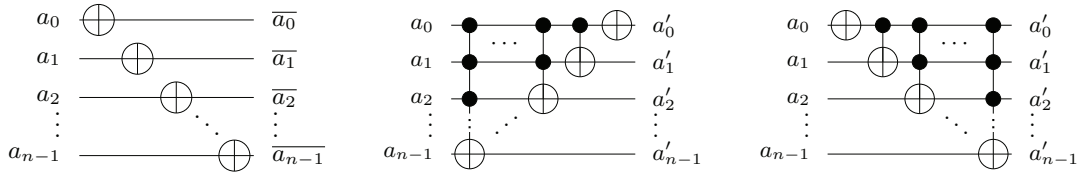
7.1.1. Bitwise Negation

To realize the bitwise negation of a variable, e.g. `~= a`, a NOT gate is added to each bit as shown in Fig. 7.1a.

7.1.2. Increment and Decrement

The increment statement describes the increase of a variable by 1 (modulo 2^n), i.e. `++= a` is the same as `a += 1`. If 1 is added to the variable a, the value of the least significant bit a_0 will be flipped. The value of all other bits a_i $(1 \leq i \leq n-1)$ will be flipped if and only if

- a_0 has an initial value of 1, so a carry is generated, and
- all bits a_j with $1 \leq j < i$ have an initial value of 1, so the carry is propagated.



(a) ~ = a (b) ++= a (c) --= a

Figure 7.1.: Building Blocks for Unary Statements

This function can be described by the following mapping.

$$\begin{aligned} a_0 &\mapsto a_0 \oplus 1 \\ a_1 &\mapsto a_1 \oplus a_0 \\ a_2 &\mapsto a_2 \oplus a_0 \cdot a_1 \\ &\vdots \\ a_{n-1} &\mapsto a_{n-1} \oplus a_0 \cdot a_1 \cdots a_{n-2} \end{aligned}$$

A straightforward realization of this mapping is shown in Fig. 7.1b. For each bit a_i ($0 \leq i \leq n-1$), a Toffoli gate with target on a_i and controls on all a_j with $0 \leq j < i$ is added. The new value of the most significant bit a_{n-1} has to be calculated first, since the initial values of $a_{n-2}, \ldots, a_0$ are used as inputs here. For the following gates, a_{n-2} is not used as a control input, so its new value can be calculated next, and so on.

The decrement statement describes the decrease of a variable by 1 (modulo 2^n), i.e. `--= a` is the same as `a -= 1`. Due to the modulo operation, a sequence of statements like `--= a; ++= a` or `++= a; --= a` will always result in a having its original value. In other words, decrement is the inverse function of increment. Let $INC = g_1 g_2 \ldots g_{d-1} g_d$ be the cascade of gates used to realize the increment statement. With this, the decrement statement can be realized as $DEC = INC^{-1} = g_d^{-1} g_{d-1}^{-1} \ldots g_2^{-1} g_1^{-1} = g_d g_{d-1} \ldots g_2 g_1$, since Toffoli gates are self-inverse. Fig. 7.1c shows this realization.

7.2. Reversible Assignments

All assignments in SyReC have the form `a` $\oplus =$ `b`, where $\oplus$ can be ^ (XOR), + (increase), or − (decrease). The right hand side b can be an arbitrary expression, possibly containing various binary operations. Sec. 7.3 describes the building blocks used to realize these expressions. Assuming the result of b is already available on given lines in the circuit, the reversible assignment of b to the left hand side a is realized by the building blocks described in this section. In all cases, the value of a is changed while the value of b needs to be preserved.

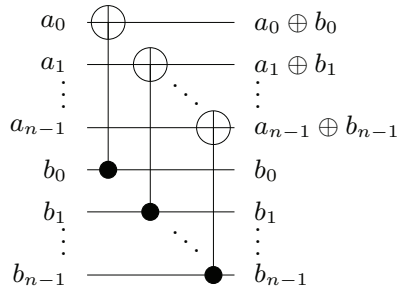


Figure 7.2.: Building Block for `a ^= b`

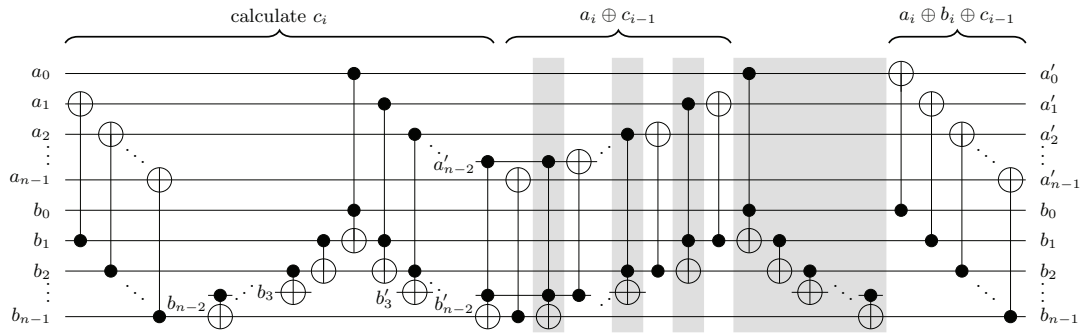


Figure 7.3.: Building Block for `a += b`

7.2.1. XOR Assignment

The XOR assignment is specified as a bitwise XOR of the left hand side and the right hand side. For example, the statement `a ^= b` describes the mapping $(a_i, b_i) \mapsto (a_i \oplus b_i, b_i)$ for all bits a_i in a and all bits b_i in b.

A circuit realization of this function is easily found. The CNOT gate flips the value on its target line if and only if the value on its control line is 1, which is equal to an XOR of the target and control line values. Thus, the desired function is realized by adding a CNOT gate for each bit of the input variables, as shown in Fig. 7.2. The value of b is passed unaltered through the controls, while the value of a is changed according to the function.

7.2.2. Increase and Decrease

The increase assignment is an addition of the right hand side to the left hand side (modulo 2^n). In the statement `a += b`, the value of b is added to the value of a. This means that the new value of each bit a_i in a can be calculated by $a_i \oplus b_i \oplus c_{i-1}$, where c_{i-1} is the carry generated by adding $a_{i-1} \dots a_0$ and $b_{i-1} \dots b_0$. If the concepts of conventional adder circuits were applied here, additional lines would have to be added for calculating and processing intermediate values like the carry for each bit. To keep the number of lines as small as possible, the values are calculated directly from the inputs, which leads

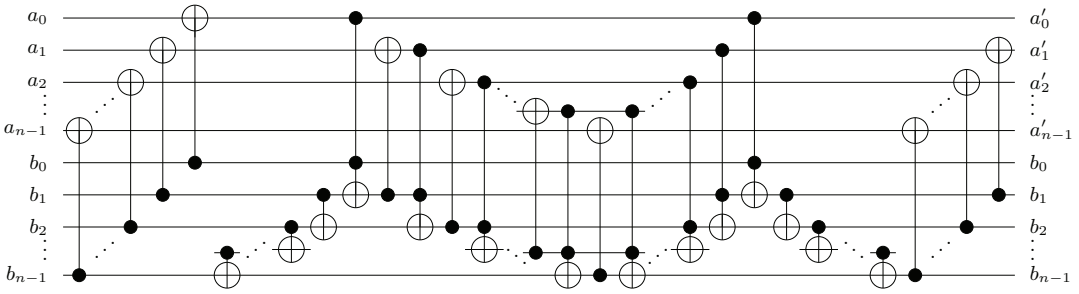


Figure 7.4.: Building Block for `a -= b`

to the following mapping.

$$
\begin{aligned}
&a_0 \mapsto a_0 \oplus b_0 \\
&a_1 \mapsto a_1 \oplus b_1 \oplus \underbrace{a_0 \cdot b_0}_{c_0} \\
&a_2 \mapsto a_2 \oplus b_2 \oplus \underbrace{a_1 \cdot b_1 \oplus a_0 \cdot b_0 \cdot a_1 \oplus a_0 \cdot b_0 \cdot b_1}_{c_1} \\
&\quad \vdots
\end{aligned}
$$

These expressions could directly be translated to a cascade of Toffoli gates: In the expression calculating $a_i \oplus b_i \oplus c_{i-1}$, each term except a_i is represented by a Toffoli gate with target on a_i and controls on all variables in the term. This representation would, however, lead to an exponentially growing number of gates needed for each bit.

Fig. 7.3 shows the proposed realization, which is much more efficient in terms of gate count and gate cost. The first part of the circuit calculates the carry for each bit. Next, the carry values are XOR-assigned to the corresponding lines of a, which are thereby set to the value $a'_i = a_i \oplus c_{i-1}$. The lines of b are simultaneously restored to their original values (highlighted in grey). Finally, the value of b is XOR-assigned to the lines of a', resulting in the required value of $a_i \oplus b_i \oplus c_{i-1}$.

The decrease assignment, e.g. `a -= b`, is a substraction of the right hand side from the left hand side (modulo 2^n). Analogous to increment and decrement (Sec. 7.1.2), the decrease assignment is the inverse function of the increase assignment. Consequently, it can be realized by using the increase building block in reverse gate order, as shown in Fig. 7.4.

7.3. Expression Operations

In SyReC, expressions can be used for calculating values to be assigned to variables, like in the statement `c ^= (a * b)`. These expressions mostly consist of bitwise, arithmetic, and shifting operations. Another use of expressions is the specification of conditions

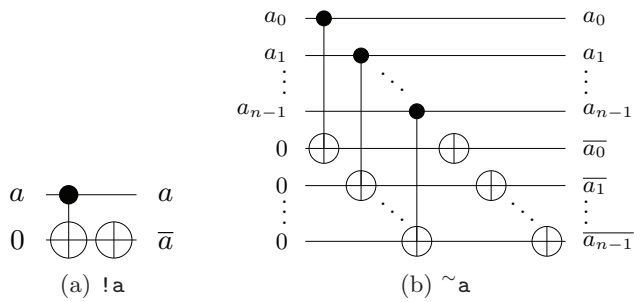


Figure 7.5.: Building Blocks for Logical and Bitwise Negation

for if-statements, like `if (a < b)`, where mostly relational and logical operations are employed. The building blocks used to synthesize expressions of any kind are described in this section. In all cases, the values of the input variables must be preserved, so new lines (with constant input value 0) are added to the circuit to calculate the output of each operation.

7.3.1. Logical and Bitwise Operations

Logical operations are designed for single bit inputs, where 1 is interpreted as *true* and 0 is interpreted as *false*. Bitwise operations are an extension of logical operations to variables with arbitrary bit widths. Here, the respective operation is performed independently on each bit, which makes the building blocks easy to expand.

Negation

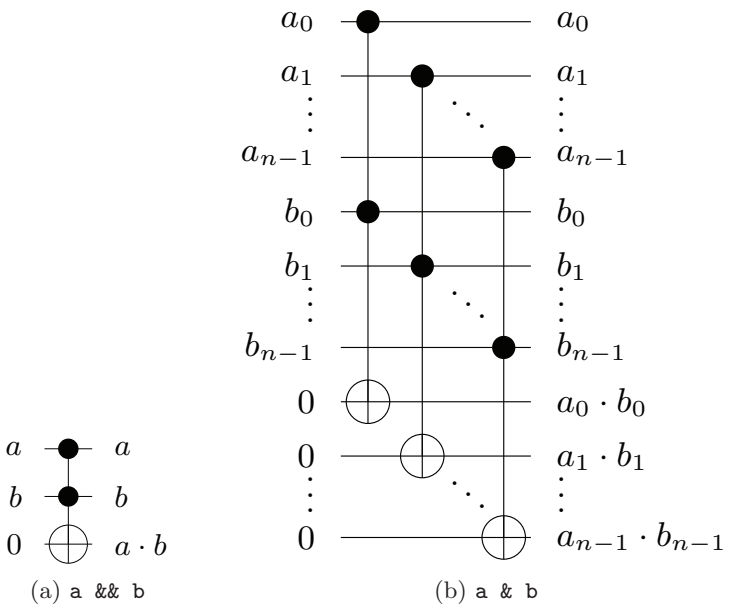
The negation of a single bit is easily accomplished by adding a NOT gate. However, since the input values need to be preserved, the original values are "copied" on new lines via CNOT gates before the actual negation is done. Fig. 7.5 shows the logical and bitwise negation.

Conjunction

The conjunction of two bits a_i and b_i can be computed with a single Toffoli gate. Its control lines are a_i and b_i, and its target line is a new line with constant input value 0. The output on the target line is $a_i \cdot b_i \oplus 0$, which is equal to $a_i \cdot b_i$. Fig. 7.6 shows the logical and bitwise conjunction.

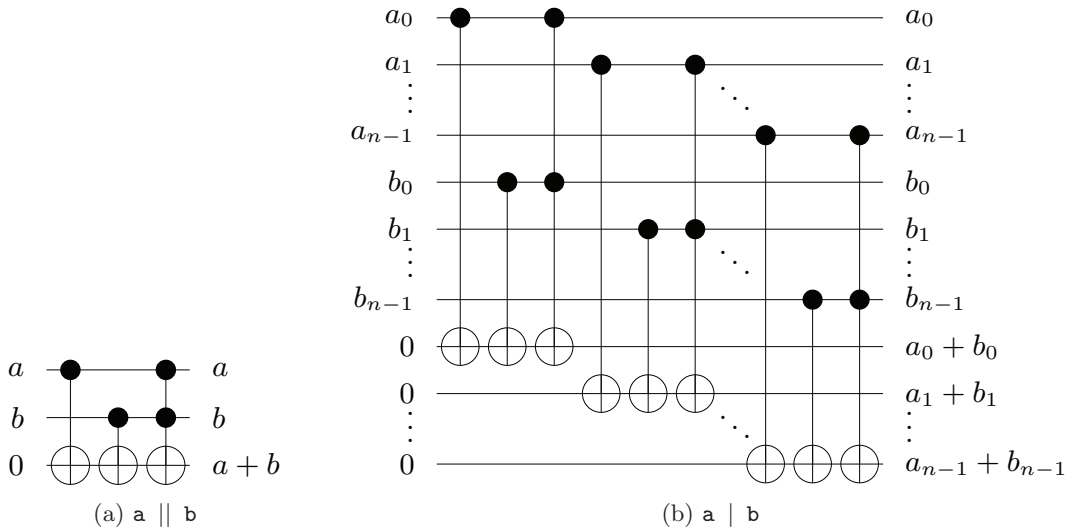
Disjunction

To realize the disjunction of two bits a_i and b_i, three gates are necessary. Since $a_i + b_i$ can not be directly translated to Toffoli gates, the equivalent function $a_i \oplus b_i \oplus a_i \cdot b_i$ is implemented. Fig. 7.7 shows the logical and bitwise disjunction.



(a) `a && b` (b) `a & b`

Figure 7.6.: Building Blocks for Logical and Bitwise Conjunction



(a) `a || b` (b) `a | b`

Figure 7.7.: Building Blocks for Logical and Bitwise Disjunction

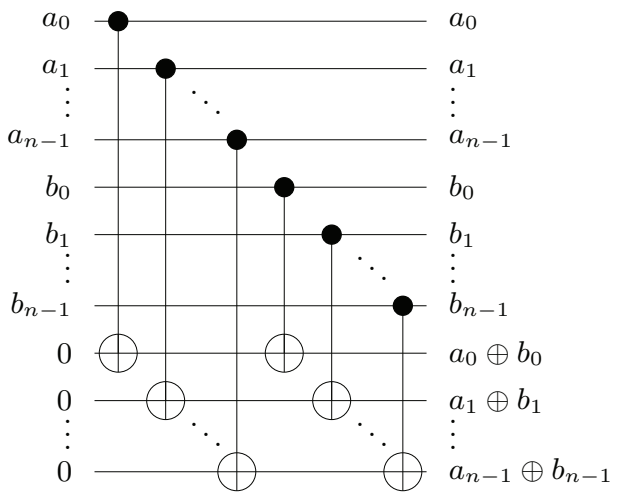


Figure 7.8.: Building Block for `a ^ b`

Exclusive Or

To compute the exclusive or of two bits a_i and b_i, the value of a_i is first "copied" to a new line using a CNOT gate, so the input value of a_i can be preserved. Then, the exclusive or is realized by adding another CNOT gate with control on b_i and target on the new line, resulting in the output $a_i \oplus b_i$. Fig. 7.8 shows the bitwise exclusive or.

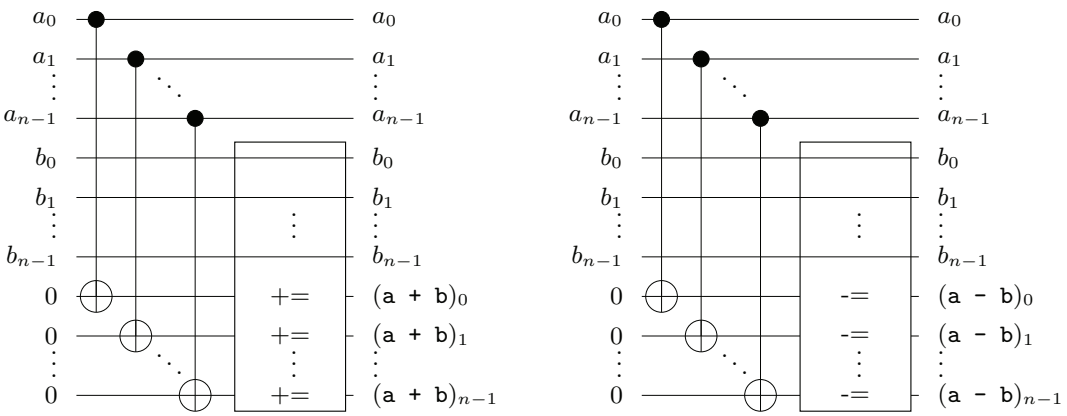
7.3.2. Arithmetic Operations

Arithmetic operations tend to be the most complicated operations, especially regarding the extension to arbitrary bit widths of the input variables. Many realizations of adders, multipliers, etc. in reversible logic have been proposed, but most of them are limited to certain bit widths (see e.g. [GW15, ME13, GK14]). The building blocks proposed here are far from optimal regarding gate cost, but automatically expandable to any bit width.

Addition and Subtraction

To calculate the expression `a + b`, a set of new lines z with constant input value 0 is added to the circuit. Then, the addition can be done by "copying" the value of a to the new lines and increasing it by b, which is equivalent to executing the statements `z ^= a; z += b`. To realize this statements, the building blocks for the assignments described in Sec. 7.2 can be used. As a result, the building block for the addition is composed of the building blocks for XOR assignment and increase, as shown in Fig. 7.9a.

The subtraction is implemented in a similar fashion, as shown in Fig. 7.9b. For `a - b`, the value of a is first "copied" and then decreased by b.



(a) `a + b` (b) `a - b`

Figure 7.9.: Building Blocks for Addition and Subtraction

Multiplication

The multiplication is realized using partial products. For example, the expression `a * b`, with a and b having a bit width of three, is calculated as follows.

$$\begin{array}{rr} & (\ b_2\, b_1\, b_0\ ) \cdot a_0 \\ + & (\ b_2\, b_1\, b_0\ ) \cdot a_1 \\ + & (\ b_2\, b_1\, b_0\ ) \cdot a_2 \\ \hline & z_2\, z_1\, z_0 \end{array}$$

Note that the bit width of the result is the same as the inputs', so only the lower bits of the arithmetic result are provided. For calculating the upper bits of the multiplication, the expression `a *> b` is used, as described in the next section.

The building block implementing the multiplication is shown in Fig. 7.10. The first block is a controlled XOR assignment, i.e. b is assigned to z iff $a_0 = 1$. Corresponding to the first line in the example calculation, this is equal to $z = a_0 \cdot b$. The next block implements the next step by calculating $z_{n-1}z_{n-2}\ldots z_1 \mathrel{+}= a_1 \cdot b_{n-2}\ldots b_1 b_0$, which is equal to $z \mathrel{+}= a_1 \cdot (b << 1)$. Continuing this scheme, all partial products are added up to the final multiplication result.

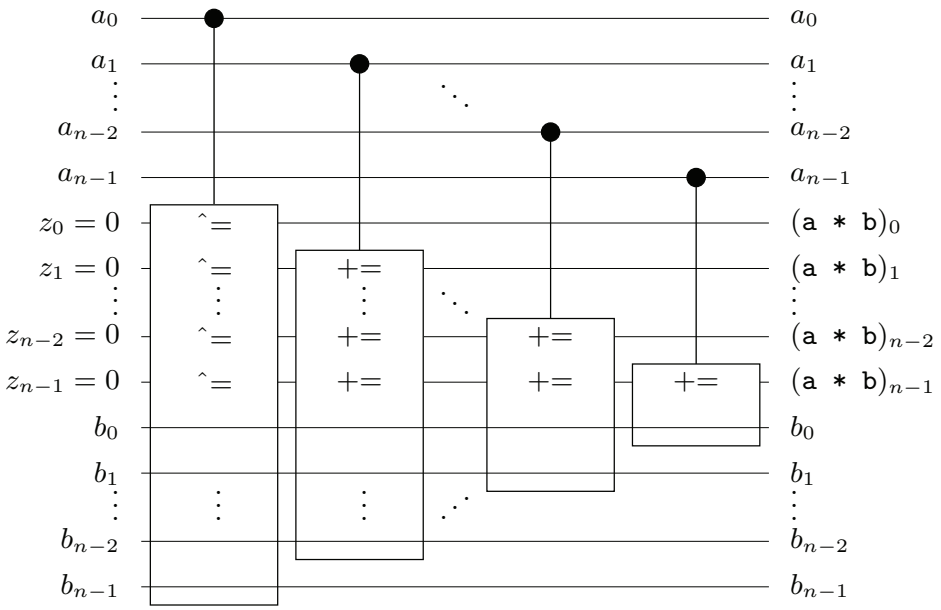


Figure 7.10.: Building Block for `a * b`

Upper Bits of Multiplication

To compute the upper bits of a multiplication, the whole product is calculated, i.e. if both input variables have a bit width of n, a set of $2n$ new lines is used to compute the result. Like in the previous section, partial products are applied, as shown in the following for a bit width of three.

$$\begin{array}{rr} & (\ b_2\ b_1\ b_0\ ) \cdot a_0 \\ + & (\ b_2\ b_1\ b_0\ )\ \ \cdot a_1 \\ + & (\ b_2\ b_1\ b_0\ )\ \ \cdot a_2 \\ \hline & \mathbf{z_5\ z_4\ z_3}\ z_2\ z_1\ z_0 \end{array}$$

Although the complete set z ist needed for the calculation, the actual result of `a *> b` is found on the "upper half" $z_5 z_4 z_3$.

Fig. 7.11 shows the implementation of the full multiplication. Again, each block calculates one partial product and adds it to the result. But since no bits are cut off here, all blocks (except the first) have the same size. For example, when computing only the lower bits of the multiplication, $z \mathrel{+}= a_1 \cdot (b << 1)$ is equal to $z_{n-1} z_{n-2} \ldots z_1 \mathrel{+}= a_1 \cdot b_{n-2} \ldots b_1 b_0$, where b_{n-1} is cut off. When computing the full multiplication, $z \mathrel{+}= a_1 \cdot (b << 1)$ is equal to $z_{n+1} z_n z_{n-1} \ldots z_1 \mathrel{+}= a_1 \cdot b_{n-1} b_{n-2} \ldots b_1 b_0$. As can be seen, one more bit of b and two more bits of z are involved. The bit z_{n+1} needs to be included in the *increase* block in case a carry is produced by adding the respective partial product. An *increase* block as described in Sec. 7.2.2 does not compute this carry value, so a slightly modified version is applied. Fig. 7.12 shows this building block, where a line and two gates (highlighted in gray) are added to calculate c_{n-1}.

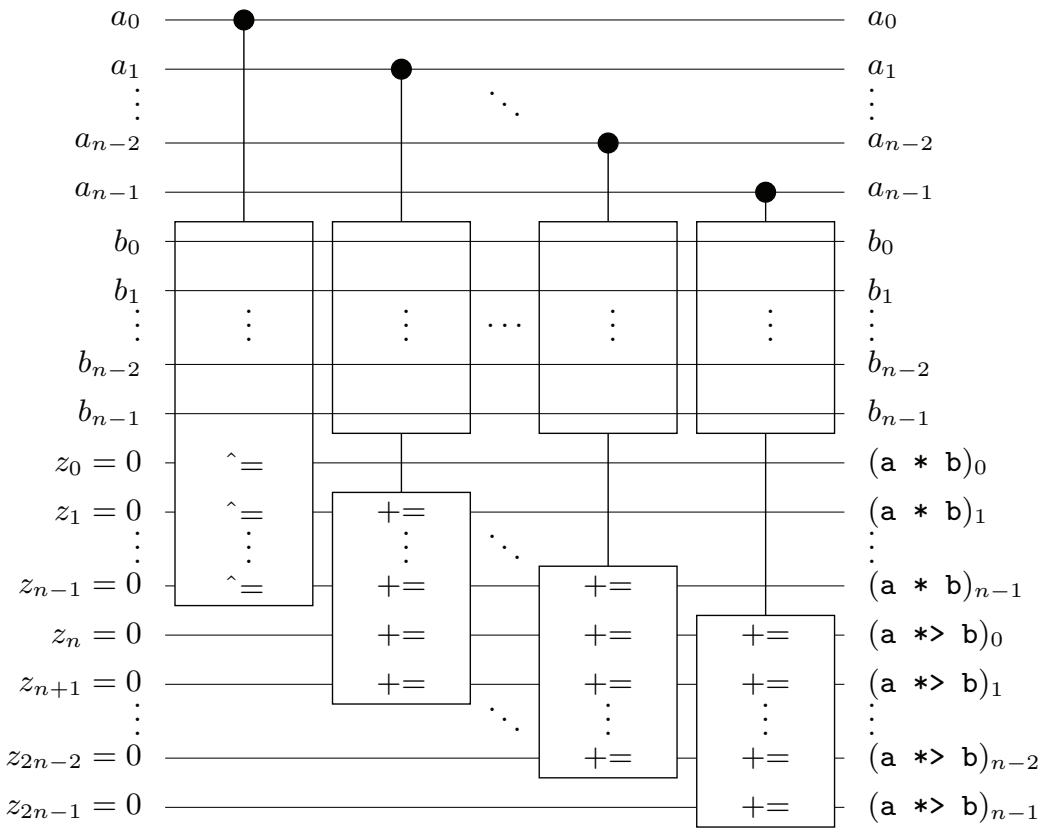


Figure 7.11.: Building Block for `a *> b`

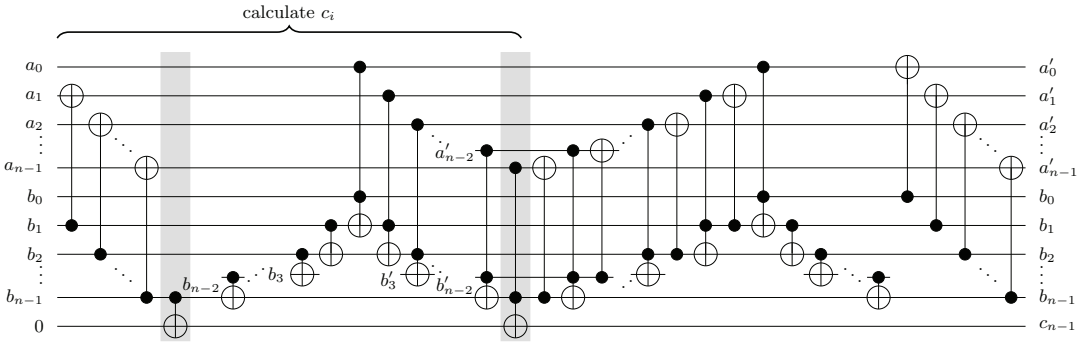
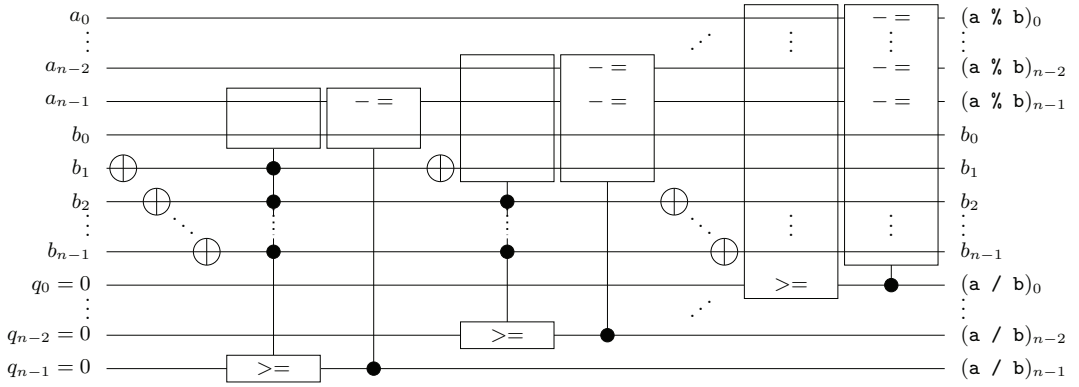
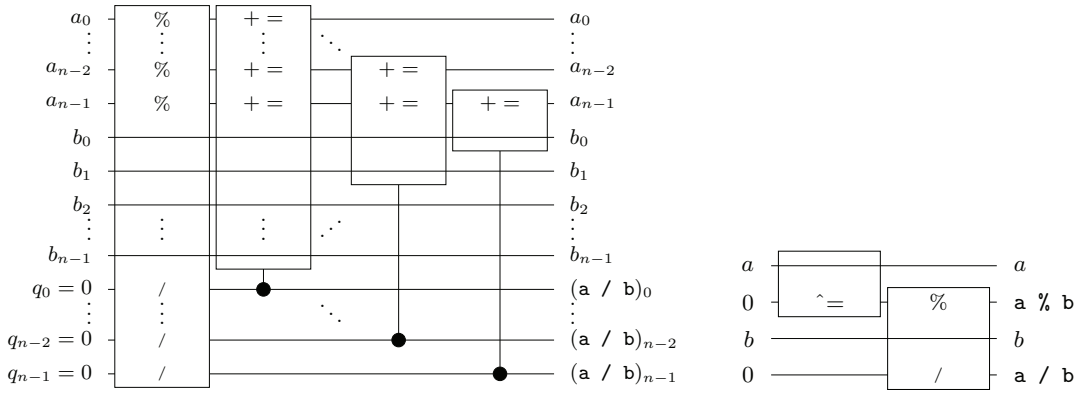


Figure 7.12.: Building Block for `a += b` with Carry Out



(a) Calculating Quotient and Remainder



(b) `a / b` (c) `a % b`

Figure 7.13.: Building Blocks for Division and Modulo

Division and Modulo

The result of a division (`a / b`) is the quotient, while the result of a modulo operation (`a % b`) is the remainder of the division. Naturally, both operations can be realized with the same building block. The underlying idea is the standard algorithm for long division, as shown in the following example for $a = 111$ and $b = 010$.

$$\begin{array}{r} 1\,1\,1 \,/\, 0\,1\,0 = 0\,1\,1 \\ -\,1\,0 \\ \hline 0\,1\,1 \\ -\,0\,1\,0 \\ \hline 0\,0\,1 \end{array}$$

First, a_2 is compared to b. Obviously, $a_2 = 1$ is smaller than $b = 10$, so b does not "go into" a_2. As a result, the first bit of the quotient is 0. Next, a_2a_1 is compared to b. Since $a_2a_1 = 11$ is not smaller than b, the second bit of the quotient is 1, and b is subtracted from a_2a_1, resulting in $a'_2a'_1$. In the last step, $a'_2a'_1a_0$ is compared to b. As b goes into

011, the third bit of the quotient is 1, and b is subtracted from $a'_2a'_1a_0$, resulting in the remainder 001.

Fig. 7.13a shows the building block implementing this algorithm. A set of new lines q with constant input value 0 is used to calculate the quotient, while the remainder is computed on the lines of a. The first step is comparing a_{n-1} to b. Naturally, a_{n-1} can only be greater than or equal to b, if $b_{n-1}b_{n-2}\dots b_2b_1 = 0$. For this reason, NOT gates are added to the lines of $b_{n-1}b_{n-2}\dots b_2b_1$ and used as control lines for the comparison of a_{n-1} and b_0. So iff $b_{n-1}b_{n-2}\dots b_2b_1 = 0$ and $a_{n-1} >= b_0$, i.e. b goes into a_{n-1}, the value of q_{n-1} is set to 1 and b_0 is subtracted from a_{n-1} (which is equal to subtracting b). Otherwise, q_{n-1} remains 0. A NOT gate is added to restore the original value of b_1 for the next step. Then, iff $b_{n-1}b_{n-2}\dots b_2 = 0$ and $a'_{n-1}a_{n-2} >= b_1b_0$, the value of q_{n-2} is set to 1 and b_1b_0 (equal to b) is subtracted from $a'_{n-1}a_{n-2}$. This scheme is continued until b is compared to and possibly subtracted from $a'_{n-1}a'_{n-2}\dots a'_1a_0$, which completes the calculation.

While this building block computes both the quotient and the remainder, it does not preserve the input value of a. Thus, additional effort is necessary for the desired implementation of the division and modulo operations. Fig. 7.13b shows the building block used to compute `a / b`. Here, the original value of a is restored by adding b to the remainder where it was subtracted before, i.e. reversing the subtractions.

The implementation of `a % b` is shown in Fig. 7.13c. Since the new value of a is the result here, the original value is preserved by "copying" it to another set of new lines before the calculation is done.

7.3.3. Shifting Operations

The shifting operations in SyReC allow a logical shift of a variable by a constant number of bit positions, either to the left or to the right. Vacant bit positions are filled up with 0. Since the shift amount is a constant value known during synthesis, the realization is straightforward.

For example, the expression `a << x` describes the following mapping.

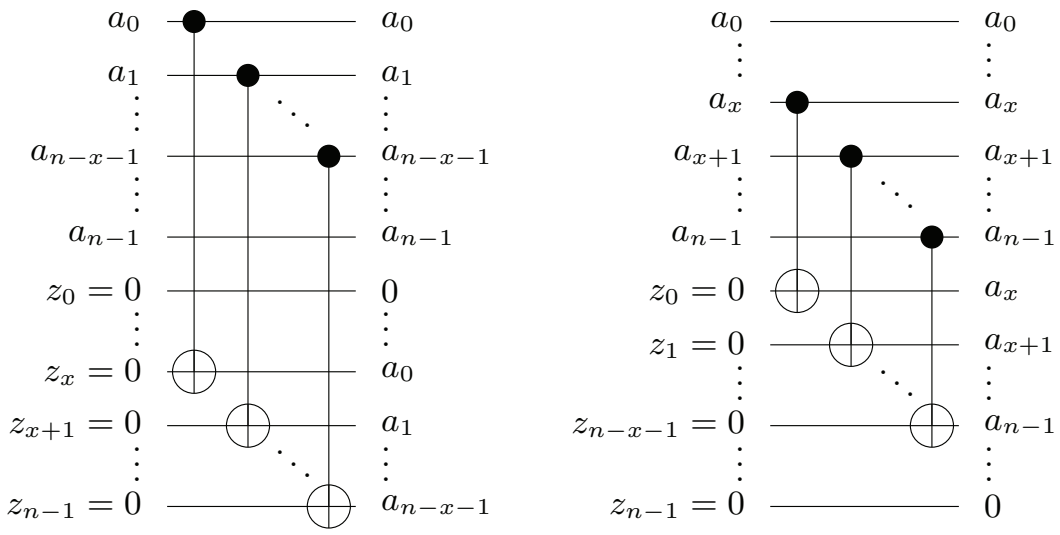
$$a_{n-1}\dots a_{n-x}a_{n-x-1}\dots a_1a_0 \;\mapsto\; a_{n-x-1}\dots a_1a_00\dots 0$$

To calculate the result, a set of new lines z with constant input value 0 is added to the circuit. The values of a_i are "copied" to the corresponding lines of z via CNOT gates, as illustrated in Fig. 7.14a. For instance, a_0 is connected to z_x, a_1 to z_{x+1}, and a_{n-x-1} to z_{n-1}. As a_{n-x} to a_{n-1} are cut off by the shifting, they are not connected to any result lines. The vacant positions z_0 to z_x keep their constant input value 0.

The right shift `a >> x` describes the mapping

$$a_{n-1}\dots a_{x+1}a_x\dots a_1a_0 \;\mapsto\; 0\dots 0a_{n-1}\dots a_{x+1}a_x.$$

Analogous to the left shift, this operation is realized with CNOT gates as shown in Fig. 7.14b.



(a) `a << x` (b) `a >> x`

Figure 7.14.: Building Blocks for Shifting Left and Right

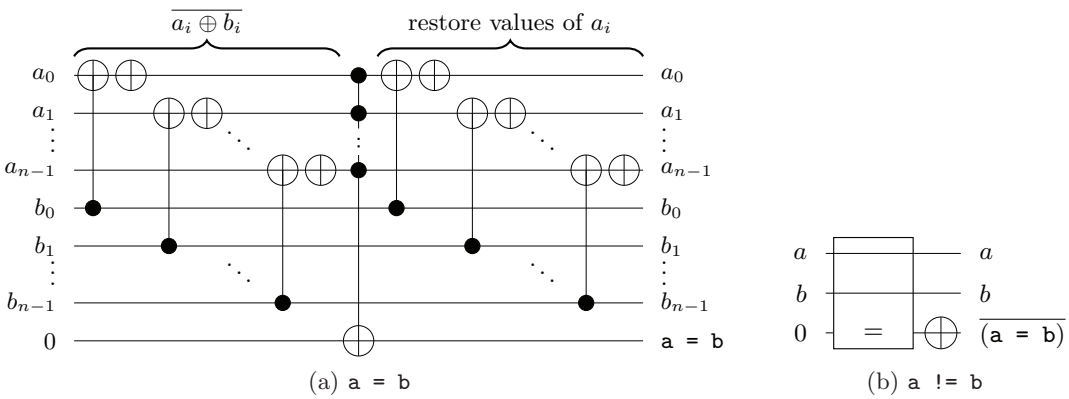
7.3.4. Relational Operations

Relational operations are used to find out e.g. if two variables have the same value or if the value of one variable is greater than the other. The input variables can be of arbitrary bit width, while the result is always just one bit (1 meaning *true* and 0 meaning *false*). Since most relations can be defined in terms of each other, like $a > b \ := \ b < a$, we decided to implement a building block for each the *equals* and *less than* operation and reduce all other relational operations to these two.

Equals and Not Equals

To determine whether two variables are equal, a bitwise comparison is implemented. The expression `a = b` should evaluate to *true* iff for all bits a_i in a and all bits b_i in b : $a_i = b_i$. First, each pair of bits a_i, b_i is compared using a CNOT and a NOT gate as shown in Fig. 7.15a, calculating $\overline{a_i \oplus b_i}$, which is equal to $a_i = b_i$. Next, a Toffoli gate with controls on all these values is applied to realize the conjunction. Its target line, the result line with constant input value 0, will only be flipped to 1, if $\overline{a_i \oplus b_i}$ is true for all i. Finally, the original value of a is restored.

For determining whether two variables are not equal to each other, the *equals* building block is used in combination with a NOT gate to negate its result. This realization is sketched in Fig. 7.15b.



(a) `a = b` (b) `a != b`

Figure 7.15.: Building Blocks for Equals and Not Equals

Less/Greater and Less/Greater or Equal

Since all variables in SyReC are interpreted as unsigned integers, the value of a variable a is less than the value of a variable b, iff

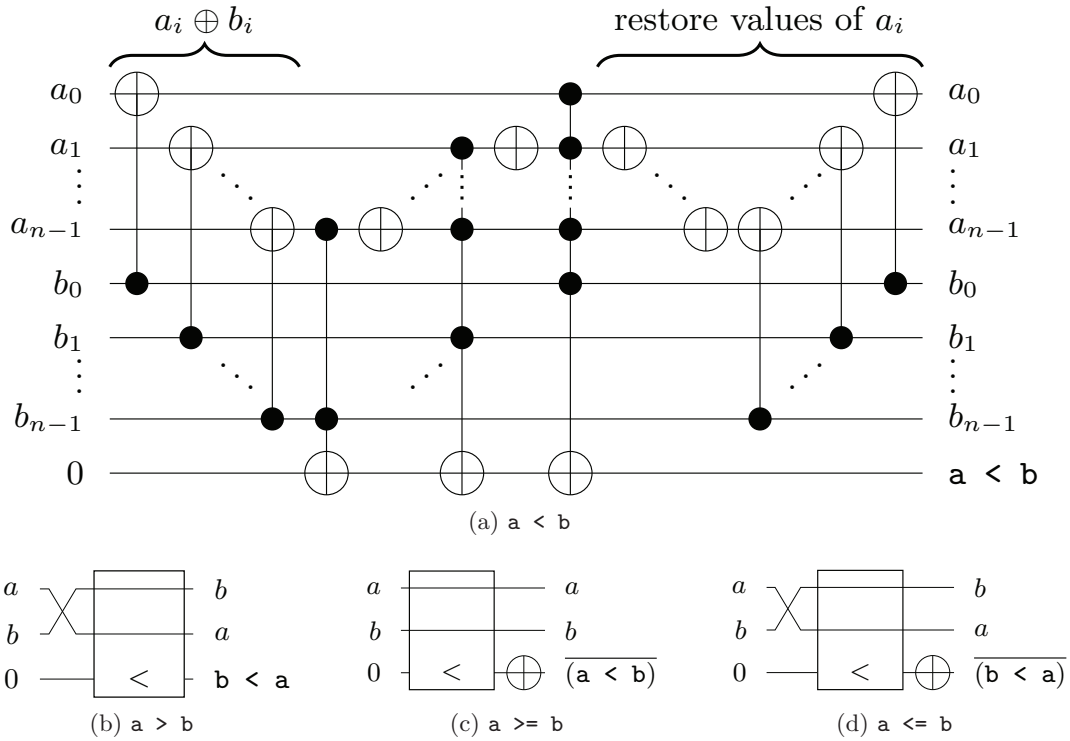
- $a_{n-1} < b_{n-1}$, i.e. the MSB of a is 0 and the MSB of b is 1, or
- $a_{n-1} = b_{n-1}$ and $a_{n-2} < b_{n-2}$, or

  $\vdots$
- $a_{n-1}a_{n-2}\ldots a_2 = b_{n-1}b_{n-2}\ldots b_2$ and $a_1 < b_1$, or
- $a_{n-1}a_{n-2}\ldots a_1 = b_{n-1}b_{n-2}\ldots b_1$ and $a_0 < b_0$.

This comparison is implemented as shown in Fig. 7.16a. First, a CNOT gate for each bit of the variables computes $a_i \oplus b_i$. Then, a Toffoli gate is applied which flips the value 0 on the result line iff $(a_{n-1} \oplus b_{n-1}) \cdot b_{n-1}$. This condition is fulfilled only if $b_{n-1} = 1$ and $a_{n-1} = 0$, i.e. $a_{n-1} < b_{n-1}$, so the Toffoli gate computes the first listed case for `a < b`. For the next case, a NOT gate is added to the line containing $a_{n-1} \oplus b_{n-1}$, thus changing its value to $a_{n-1} = b_{n-1}$. Since this is a condition for all following cases of `a < b`, the respective line is used as a control line for all following Toffoli gates. The next case, i.e. $a_{n-1} = b_{n-1}$ and $a_{n-2} < b_{n-2}$, is represented by a Toffoli gate with controls on $a_{n-1} = b_{n-1}$, $a_{n-2} \oplus b_{n-2}$, and b_{n-2}. After that, another NOT gate is used to introduce the condition $a_{n-2} = b_{n-2}$ for all following cases. This scheme is continued until each case for `a < b` is represented by a Toffoli gate. Since all cases are exclusive, the value on the result line will be flipped to 1 by exactly one Toffoli gate if `a < b` is true, and remain 0 otherwise.

After the computation of `a < b`, the lines of a are restored to their original values.

To compute `a > b`, which is equal to `b < a`, the *less than* building block is applied with switched inputs, as sketched in Fig. 7.16b.



(a) `a < b`

(b) `a > b`

(c) `a >= b`

(d) `a <= b`

Figure 7.16.: Building Blocks for Less/Greater and Less/Greater or Equal

The operation `a >= b` can be defined as $\overline{(a < b)}$. Consequently, it is implemented using the *less than* block in combination with a NOT gate, as shown in Fig. 7.16c.

In a similar way, `a <= b` can be defined as $\overline{(b < a)}$. Fig. 7.16d shows the implementation using the *less than* building block with switched inputs and negating its result.

7.4. Conclusion

In this chapter, we gave a detailed description of the building blocks used in SyReC synthesis. Since these circuit patterns have to be expandable to arbitrary bit widths of their input variables, their realizations are handmade, some with a large overhead in terms of gate count and cost. While not all additional expenses can be avoided, there surely is potential for optimization in some building blocks. When a more efficient pattern is found to realize an operation, the corresponding building block can easily be replaced.

8. Optimization of SyReC Synthesis

In this chapter, we present two approaches to optimize the synthesis of SyReC specifications. First, an extended synthesis scheme is presented, which can reduce the number of circuit lines using additional gates. Second, a scheme is presented which can reduce the gate cost of resulting circuits, using one additional circuit line. This method can be applied to both the original and the extended synthesis scheme. After presenting the optimization methods, Section 8.3 provides an experimental evaluation. Here, the initial synthesis scheme is compared to a BDD-based synthesis scheme first before investigating the effects of the line- and cost-aware synthesis.

8.1. Line-aware Synthesis

In order to realize SyReC specifications with a smaller number of additional circuit lines, an extended synthesis scheme is presented in this section (based on [WSSD12]). The idea is to use the same building blocks as introduced in the previous chapters, but to undo intermediate results of the expressions as soon as they are not needed anymore. A similar idea (for reversible software programs) has previously been proposed in [Axe11]. This enables that circuit lines which have been occupied by expressions before can be re-used.

In the following, the general concept of this scheme is illustrated before the extended synthesis is described in detail for all possible SyReC statements. Afterwards, the necessary amount of additional circuit lines is discussed.

8.1.1. General Concept

The extended synthesis approach follows the scheme as introduced in Chapter 6, but is extended by an additional third step:

3) Add the inverse circuit from Step 1, i.e. $G_{\odot}^{-1}$, to the circuit in order to reset the circuit lines buffering the result of the expressions to the constant 0.

Example 10 *Consider the two following generic HDL statements:*

```
a ⊕= (b ⊙ c);
d ⊕= (e ⊙ f);
```

Fig. 8.1 sketches the resulting circuit after applying the extended synthesis scheme. The first two sub-circuits $G_{b\odot c}$ and $G_{a\oplus=b\odot c}$ ensure that the first statement is realized. This is equal to the scheme proposed in Chapter 6 and leads to additional lines with constant inputs (highlighted thick). Afterwards, a further sub-circuit $G_{b\odot c}^{-1}$ is applied. Since $G_{b\odot c}^{-1}$

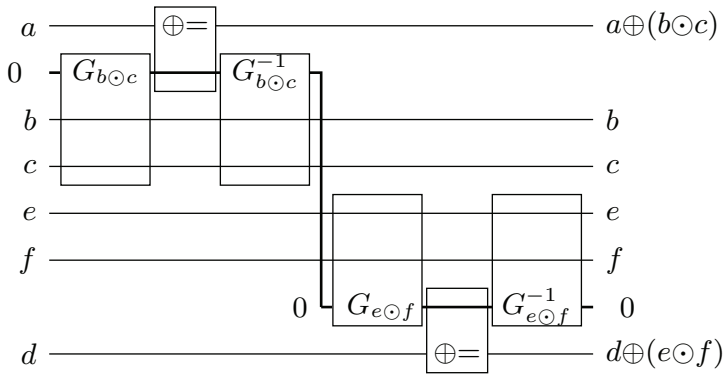


Figure 8.1.: Scheme for Line Reduction in SyReC Synthesis

is the inverse of $G_{b\odot c}$, this sets the circuit lines buffering the result of $b \odot c$ back to the constant 0. As a result, these circuit lines can be reused in order to realize the following statements as illustrated for $d\oplus{=}e \odot f$ in Fig. 8.1.

8.1.2. Resulting Synthesis Scheme

Following the proposed concept, each statement can be realized with zero garbage outputs. In the following, the precise realization of this scheme is detailed for each possibly affected statement. The unary statements, the swap-statement (`<=>`) and the skip-statement are not considered here as they are realized without additional circuit lines.

Assignment Statements

In order to realize statements of the form $a\oplus{=}e$ with e being an arbitrary expression, basically the respective building blocks are orchestrated as already illustrated in Fig. 8.1. First, a sub-circuit realizing the expression e, i.e. the right-hand side of the statement, is created. This requires additional lines to store the result of e. Next, a sub-circuit realizing the assignment operation is created as well as a sub-circuit reversing the result of e into a constant value. The latter is done by reversing the order of gates of the first sub-circuit. Finally, all three sub-circuits are composed leading to the desired realization of the statement.

Example 11 *Fig. 8.2 shows the circuit obtained by synthesizing $c\hat{}{=}(a+b)$ using the extended synthesis scheme. The respective sub-circuits G_{a+b}, $G_{c\hat{}=a+b}$, and G^{-1}_{a+b} are highlighted by dashed rectangles. Since all gates considered in this work are self-inverse, G^{-1}_{a+b} is obtained by reversing the order of the gates of G_{a+b}.*

Applying this procedure, any arbitrary combination of assignment statements and expressions can be realized in a garbage-free manner. That is, required additional circuit lines are ancilla lines and can be reused for other statements and operations.

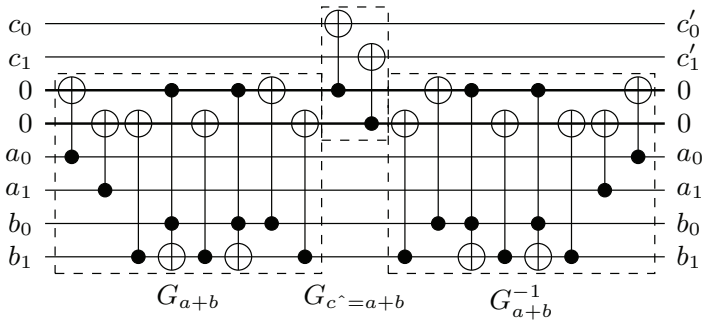


Figure 8.2.: Synthesizing `c ^= (a+b)`

Conditional Statements

As described in Section 6.4.2, there are two proposed realizations for conditional statements (cf. Fig. 6.4).

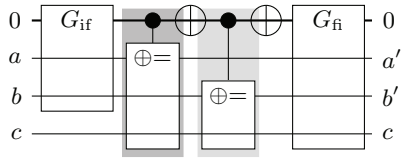
Fig. 8.3b illustrates the adjusted procedure for the synthesis of a conditional statement according to the first realization (i.e. according to the scheme illustrated in Fig. 6.4b). The gates needed to realize the *then*-block (*else*-block) are highlighted in dark gray (light gray). Also here, a sub-circuit G_{if} evaluating the respective *if*-expres-sion is created. The intermediate results of that expression are handled analogously to assignment statements as described above. An additional circuit line is applied to store the Boolean result of the *if*-expression and control the execution of the *then*- and *else*-block as described in Section 6.4.2. The flip on the additional line, which is done to control the gates of the *else*-block, is then restored by another NOT gate. Afterwards, the original (constant) value of that line is restored by applying a sub-circuit G_{fi} which evaluates the *fi*-expression of the statement analogous to G_{if}. As defined in Section 5.3.3, SyReC requires the definition of a *fi*-expression that evaluates to the same Boolean value as the *if*-expression did in G_{if}.

Besides that, Fig. 8.3c illustrates the adjusted procedure for the synthesis of a conditional statement according to the second realization (i.e. according to the scheme illustrated in Fig. 6.4c). The gates highlighted in dark gray (light gray) correspond to the *then*-block (*else*-block). Also here, a sub-circuit G_{if} is created as in the first realization, and the result of the *if*-expression is stored in an additional line e (the top line in Fig. 8.3c). The conditional statement is then realized by applying the procedure described in Section 6.4.2. Afterwards, the values of the additional lines that were used to duplicate signals are reset to the constant value 0. This is done by applying the gates used in the *then*- and *else*-block again with e as an additional control line. The additional lines are set to the values of the corresponding signal lines, which are then used to undo the duplication and set the additional lines back to 0. The value of e is reset to 0 by creating a sub-circuit G_{fi} as in the first realization.

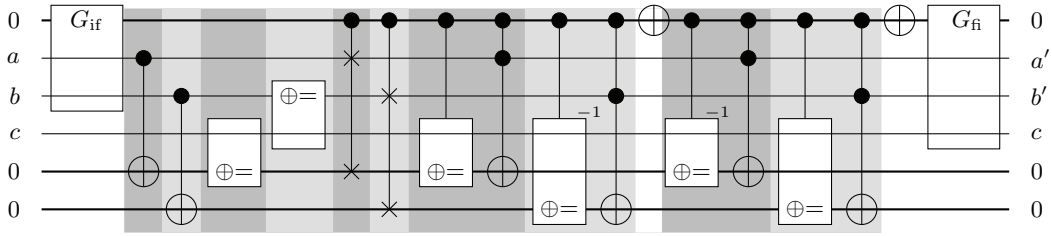
The original advantage of the second realization was lower quantum cost and transistor cost, since the realization of the *then*- and *else*-block does not have an extra control line on each gate. This advantage is lost here. Since the values of the additional lines depend

```
if (a = b)
 then
  a ⊕= c
 else
  b ⊕= c
fi (a = (b ⊕ c))
```

(a) Code



(b) Without Additional Lines



(c) With Additional Lines

Figure 8.3.: Synthesizing Conditional Statements

on the value of e (e.g. a if $e = 1$ and a' if $e = 0$) and the realizations of the *then*- and *else*-block are needed to set the additional line to the same value as the signal line (e.g. a' if $e = 1$ and a if $e = 0$), both the realization of the *then*- and *else*-block have to be added to the circuit with an extra control line on each gate. As a consequence, the second realization of conditional statements in the line-aware synthesis leads to both, additional circuit lines as well as higher costs, and is therefore not considered any further.

Loops and Calls

The realization of loops and module calls is treated in a straight forward manner exploiting the procedures proposed above. More precisely, calls are substituted by the corresponding statements inside the body of the call. Loops are realized by explicitly cascading (i.e. unrolling) the respective statements within a loop block according to the fixed and finite number of iterations.

8.1.3. Discussion

Applying the extended synthesis scheme, every statement is synthesized with zero garbage outputs and only additional ancilla lines. Consequently, the total number of additional lines which are required to realize a SyReC specification with the proposed solution can be determined by the statement that requires the largest number of additional lines in order to buffer intermediate results.

Example 12 *Consider a sequence of three assignment statements to be synthesized. Additionally, assume that 1, 3, and 2 circuit lines are needed to buffer the intermediate results of the respective expressions. Then, in total* $\max\{1, 3, 2\} = 3$ *additional circuit lines are needed to realize the statements. Fig. 8.4 illustrates how these circuit lines*

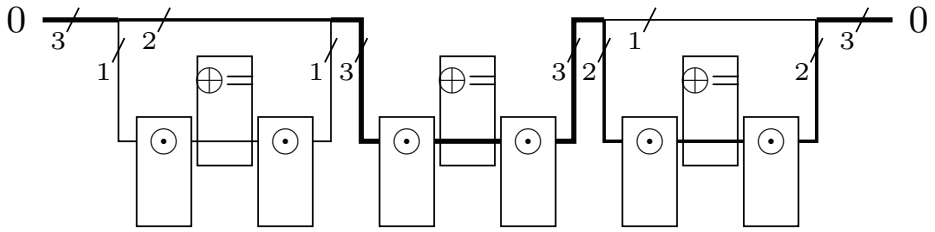


Figure 8.4.: Effect of Expression Size on Resulting Circuit

are applied. For comparison, the synthesis scheme from Chapter 6 needs $1 + 3 + 2 = 6$ *additional circuit lines.*

The number of additional circuit lines can further be reduced in many cases by restructuring the SyReC code. In general, larger expressions lead to more intermediate results to be buffered. Thus, if the same functionality can be represented by more but smaller statements, a further reduction in the number of lines is possible.

Example 13 *Consider the following statement:*

```
a += ((b & c) + ((d * e) - f))
```

In order to execute the outer expression (i.e. the addition operation), the intermediate results of the inner expressions `(b & c)`*,* `(d * e)`*, and* `((d * e) - f)` *are buffered at the same time. Considering 32-bit signals, this requires* 96 *circuit lines (in addition to* 32 *circuit lines needed to buffer the result of the outer expression itself, i.e.* 128 *in total).*

In contrast, the same functionality can also be specified by the following statements.

```
a += (b & c);
a += (d * e);
a -= f;
```

Here, the respective binary operations are applied separately with an assignment operation. Hence, no more than 32 *ancilla lines are needed to buffer the intermediate results.*

Overall, a price for the smaller number of circuit lines is an expected increase in the number of gates, and thus in the gate costs. However, the increase in the gate costs is bounded. For example, in comparison to the synthesis scheme from Chapter 6 where the building blocks $G_{\odot}$ and $G_{\oplus}$ are applied for each assignment statement, the extended scheme uses just one more building block $G_{\odot}^{-1}$. Since $G_{\odot}^{-1}$ is the inverse of $G_{\odot}$, the circuit can at most double its gate cost.

Overall, the resulting circuits still include additional circuit lines with constant inputs. But considering that, until today, the synthesis of complex functionality as a reversible circuit with the minimal number of lines is a cumbersome task (e.g. [WKD11]), the proposed solution enables to keep this number relatively small.

8.2. Cost-aware Synthesis of SyReC Specifications

Finally, all synthesis approaches proposed in the previous sections can further be refined in order to reduce the costs of the resulting circuits. An observation made in [MWD10] is exploited for this purpose. Here, it has been observed that many reversible circuits are composed of cascades of gates with several common control lines. As reviewed in Section X, the costs of single gates mainly depend on their respective number of control lines. Hence, buffering the results of common control conditions of a cascade of gates enables to reduce the number of required control lines in each gate. As a result, the costs of each gate and, hence, the costs of the entire circuit are decreased significantly.

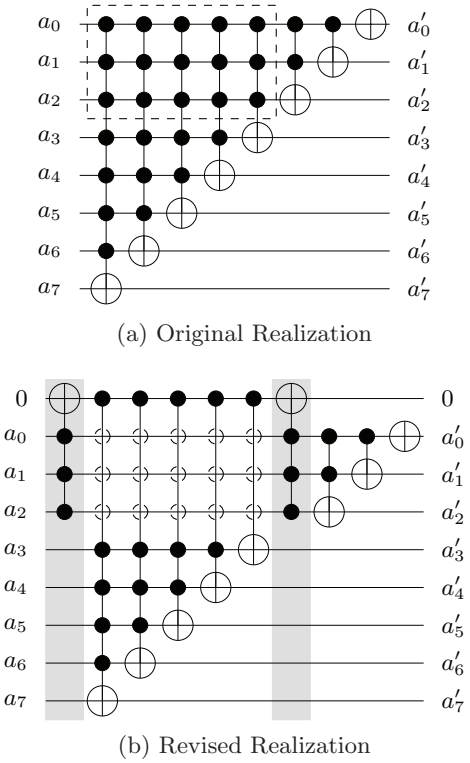
Example 14 *Consider an 8-bit realization of the increment statement (++=a) as shown in Fig. 8.5a. The gates in this cascade have several common control lines, e.g.* $C' = \{a_0, a_1, a_2\}$. *By adding two Toffoli gates* $\mathrm{T}(C', \{h\})$, *the result of the common control conditions* C' *can be buffered in an ancilla line* h *as shown in Fig. 8.5b (the new gates are emphasized with a gray box and the line* h *is on the top). This enables that all gates with control lines* $C \supseteq C'$ *can be simplified, i.e. instead of* C *a smaller set of control lines* $(C \setminus C') \cup \{h\}$ *is sufficient (in Fig. 8.5b, the saved control lines are indicated with dashed circles). As a result, the costs of the gates and, hence, the costs of the overall circuit are significantly reduced. In fact, quantum costs can be improved from 431 to 116 (73%) and transistor costs can be improved from 224 to 192 (14%).*

Similar observations can be made for many other building blocks as well. Particularly (nested) conditional statements frequently lead to large cascades of gates with common control lines. This is because the circuit lines representing the conditional expressions control whole cascades realizing the respective *then*- and *else*-blocks. Hence, it is worth to exploit these observations.

Note that an improvement is obviously only possible if the total costs of the two added gates are less than the costs saved by buffering the common control lines. Furthermore, a free ancilla line has to be available. This is either already the case (e.g. when a constant circuit line is required anyway for the realization of another expression) or can explicitly be added by the designer to enable this reduction.

Following these concepts, synthesis of SyReC specifications can be refined as follows:

1. Synthesize a statement as described in the previous sections.
2. Determine cascades of gates $t_1(C_1, T_1) \dots t_k(C_k, T_k)$ which satisfy the following criteria:
   a) The gates in the cascade have a common set C' of control lines, i.e. $C_i \supseteq C'$ for $1 \leq i \leq k$.
   b) The value of the common control lines is not modified within this cascade, i.e. $C' \cap T_i = \emptyset$ for $1 \leq i \leq k$.
3. Create a new cascade $\mathrm{T}(C', \{h\})t_1((C_1 \setminus C') \cup \{h\}, T_1) \dots t_k((C_{k+1} \setminus C') \cup \{h\}, T_k)\mathrm{T}(C', \{h\})$.



(a) Original Realization

(b) Revised Realization

Figure 8.5.: Scheme for Cost Reduction in SyReC Synthesis

4. If a free circuit line h is available and the new cascade is cheaper than the original cascade, replace the original cascade with the new one.

This procedure is applicable to both synthesis approaches, i.e. to the scheme proposed in Chapter 6 as well as to the extended scheme proposed in Section 8.1. Determining the best possible cascades for replacement is a complex task as the order in which common control lines are exploited typically has an effect. Hence, we apply this procedure only for single statements leading to local optima. As confirmed by the experiments in the next section, this leads to significant improvements in short run-time.

8.3. Evaluation of the Resulting Circuits

Besides the case study on the applicability of the hardware description language in Chapter 9, we also conducted a thorough study on the quality of the resulting circuits. For this purpose, we implemented all synthesis schemes as described above in C++ on top of *RevKit* [SFWD12]. As benchmarks for the evaluation, we used the SyReC specifications from the respective CPU components discussed in the previous section as well as further designs which have been made available at *RevLib* [WGT+08]. All experiments have been performed on a 2.8 GHz Intel Core i7 processor with 7.8 GB of main memory. In the following, the results are summarized and discussed.

Table 8.1.: Comparison of SyReC Synthesis to BDD-based Synthesis

			BDD-based Synth. [WD09]				SyReC Synth. *if* w/o add. Lines			SyReC Synth. *if* with add. Lines		
Benchmark	*bw*	PI/PO	a.L.	QC	TC	run-time	a.L.	QC	TC	a.L.	QC	TC
CPU from Chapter 9												
cpu_alu	16	55	1 852	20 660	77 704	165.99	349	662 531	568 328	2 085	31 244	67 896
cpu_alu	32	103	–	–	–	>500	653	2 235 491	1 917 448	6 101	112 396	218 680
cpu_control_unit	16	233	618	7 119	27 264	0.12	158	40 433	43 888	413	22 343	31 432
cpu_pc	11	24	39	392	1 456	0.00	13	857	912	68	797	1 336
cpu_register	16	149	512	7 040	25 600	0.05	18	9 833	8 472	162	7 560	8 472
cpu_register	32	293	1 024	14 080	51 200	0.21	34	19 641	16 792	322	15 096	16 920
Benchmarks from RevLib [WGT+08]												
alu	16	50	–	–	–	>500	67	258 872	234 424	115	146 385	151 168
alu	32	98	–	–	–	>500	131	1 704 912	1 402 232	227	1 230 577	1 064 000
alu_flat	16	50	–	–	–	>500	68	181 662	179 464	132	146 496	151 472
alu_flat	32	98	–	–	–	>500	132	1 380 526	1 177 928	260	1 230 784	1 064 560
simple_alu	16	50	–	–	–	>500	67	35 463	39 552	115	6 275	17 568
simple_alu	32	98	–	–	–	>500	131	144 791	154 432	227	25 531	67 744
bubblesort	16	64	–	–	–	>500	254	29 327	44 248	748	21 149	43 272
bubblesort	32	128	–	–	–	>500	494	58 739	88 840	1 468	42 281	87 096
callif	16	33	499	7 031	26 128	3.80	1	1 522	3 816	33	641	2 664
callif	32	65	–	–	–	>500	1	3 154	7 912	65	1 313	5 480
mult_stmts	16	96	–	–	–	>500	32	6 122	16 960	32	6 122	16 960
mult_stmts	32	192	–	–	–	>500	64	25 282	66 752	64	25 282	66 752
nestedif	16	34	752	10 534	39 128	11.04	3	6 982	11 000	99	1 475	5 848
nestedif	32	66	–	–	–	>500	3	14 470	22 776	195	3 011	11 992
nestedif2	16	34	257	3 348	12 312	1.72	4	8 423	8 856	100	6 034	6 824
nestedif2	32	66	–	–	–	>500	4	31 703	27 736	196	26 674	23 784
varops	16	48	–	–	–	>500	64	1 361	6 512	64	1 361	6 512
varops	32	96	–	–	–	>500	128	2 801	13 424	128	2 801	13 424

8.3.1. Comparison to Previous Work

In a first evaluation, we compared the quality of the circuits obtained using the initial synthesis scheme (as introduced in Chapter 6) to previously proposed solutions. As discussed in Chapter 1, most of the existing synthesis approaches for reversible circuits rely on non-compacted Boolean descriptions and are therefore often not scalable. In fact, the complex circuitry considered here cannot be realized by most of them. The BDD-based synthesis approach presented in [WD09] represents an exception as it relies on a compacted Boolean representation. Hence, we compared the circuits generated by SyReC with the equivalent realizations generated by the approach from [WD09].

The results are summarized in Table 8.1. The first columns give the name of the benchmark, the bit width of the realization as well as the number of primary inputs and outputs (denoted by *Benchmark*, *bw*, and *PI/PO*, respectively). The following columns give the number of additional circuit lines (*a.L.*), the quantum cost (*QC*), and the transistor cost (*TC*) of the circuits obtained using the BDD-based approach (denoted by *BBD-based synth.*) and the SyReC synthesizer. For the latter, we distinguish between the realization of if-statements according to Fig. 6.4b (denoted by *if w/o add. Lines*) and according to Fig. 6.4c (denoted by *if with add. Lines*). For the BDD-based approach the *run-time* is additionally listed. This is omitted for the SyReC solution as *all* circuits have been realized in less than one CPU second.

As can be clearly seen, the proposed approach outperforms the BDD-based synthesis with respect to scalability. In particular for the benchmarks including arithmetic (e.g. the *alu* realizations), BDD-based synthesis requires a significant amount of time to generate a result; often the results cannot be achieved within the applied timeout of 500 CPU seconds. This can be explained by the fact, that in particular for the multiplication no efficient representation as BDD exists. Thus, for these components the BDD-based approach suffers from memory explosion.

Besides that, these results also confirm the discussion from Section 6.4.2 concerning the different realizations of the *if*-statements. If additional circuit lines are applied, the respective costs can significantly be reduced. In comparison to the realization without additional circuit lines for *if*-statements, approx. 40% (95% in the best cases) of the quantum costs and more than 20% (90% in the best cases) of the transistor costs can be saved. In contrast, this leads to a significant increase in the number of additional lines.

8.3.2. Effect of Line- and Cost-aware Synthesis

In a second evaluation, the effect of the optimized synthesis schemes presented in Section 8.1 (for line-aware synthesis) and Section 8.2 (for cost-aware synthesis) has been evaluated. Here, Table 8.2 presents the results generated with the following schemes:

- The synthesis scheme as described in Section 8.1 using the realization of *if*-statements according to Fig. 6.4b (denoted by *Line-aware synth.*)[1],
- the synthesis scheme as described in Section 8.2 using the realization of *if*-statements according to Fig. 6.4b (denoted by *Cost-aware synth.;* if *w/o add. Lines*),
- the synthesis scheme as described in Section 8.2 using the realization of *if*-statements according to Fig. 6.4c (denoted by *Cost-aware synth.;* if *with add. Lines*), and
- the synthesis scheme as described in Section 8.1 and Section 8.2 combined together with the realization of if-statements according to Fig. 6.4b (denoted by *Cost-aware + Line-aware synth.*).

Beyond that, Table 8.2 uses the same denotation as Table 8.1. To further ease the interpretation of the numbers, we additionally provide the average values of the respective metrics for *all* considered synthesis schemes in Table 8.3.

The observations from above are confirmed. In fact, it becomes clearly evident that the selection of the respective scheme is crucial to the resulting circuit sizes. Differences of several orders of magnitude can be observed for all objectives. On average, the number of additional lines varies from 48.8 (if the line-aware scheme is applied) to 559.7 (if schemes are applied realizing *if*-statements according to Fig. 6.4c). Similarly, the worst case quantum costs (transistor costs) of 558,967.7 (490,699.7) can be reduced to 27,271.7 (58,410.7) if cost-aware synthesis and the realization of if-statements with additional lines

[1]Note that a realization of if-statements according to Fig. 6.4c has not been considered for this scheme since, as discussed in Section 8.1.2, line-aware synthesis would always lead to an increase in both, additional lines and costs, in this case.

Table 8.2.: Effect of Line- and Cost-aware SyReC Synthesis

		Line-aware Synth. (Sec. 8.1)			Cost-aware Synth. (Sec. 8.2)						Cost-aware + Line-aware Synth.		
					if w/o Add. Lines			*if* with Add. Lines					
Benchmark	*bw*	a.l.	QC	TC	a.l.	QC	TC	a.l.	QC	TC	a.l.	QC	TC
CPU from Chapter 9													
cpu_alu	16	87	1 281 717	1 103 200	350	63 025	112 048	2 086	30 208	67 144	88	118 751	215 568
cpu_alu	32	151	4 381 653	3 766 496	654	178 783	337 000	6 102	107 136	215 112	152	331 151	648 208
cpu_control_unit	16	57	80 142	87 176	159	10 513	23 808	414	7 463	21 448	58	20 756	47 304
cpu_pc	11	13	865	944	14	505	672	69	609	1 224	14	513	704
cpu_register	16	17	9 848	8 512	19	2 217	3 352	163	2 600	5 144	18	2 232	3 392
cpu_register	32	33	19 656	16 832	35	3 577	5 528	323	4 760	9 496	34	3 592	5 568
Benchmarks from RevLib [WGT+08]													
alu	16	19	516 628	467 184	68	44 782	81 888	116	35 152	72 008	20	88 566	162 208
alu	32	35	3 407 588	2 801 136	132	174 594	319 888	228	150 928	297 224	36	347 198	636 672
alu_flat	16	17	363 012	357 904	69	38 657	76 872	133	35 263	72 312	18	77 002	152 720
alu_flat	32	33	2 760 420	2 353 808	133	158 241	307 208	261	151 135	297 784	34	315 850	612 368
simple_alu	16	19	69 810	77 440	68	8 975	21 088	115	6 275	17 568	20	17 262	40 816
simple_alu	32	35	287 346	305 536	132	30 775	74 592	227	25 531	67 744	36	60 206	146 544
bubblesort	16	153	34 374	53 512	255	11 615	31 960	749	12 653	36 360	154	13 830	38 920
bubblesort	32	297	68 766	107 320	495	21 827	61 192	1 469	24 569	70 968	298	25 950	74 296
callif	16	1	1 524	3 824	1	1 522	3 816	33	641	2 664	1	1 524	3 824
callif	32	1	3 156	7 920	1	3 154	7 912	65	1 313	5 480	1	3 156	7 920
mult_stmts	16	16	11 572	30 704	32	6 122	16 960	32	6 122	16 960	16	11 572	30 704
mult_stmts	32	32	49 172	126 832	64	25 282	66 752	64	25 282	66 752	32	49 172	126 832
nestedif	16	2	6 996	11 056	4	3 094	7 800	99	1 475	5 848	3	3 108	7 856
nestedif	32	2	14 484	22 832	4	6 358	15 992	195	3 011	11 992	3	6 372	16 048
nestedif2	16	3	8 504	9 072	5	5 243	6 568	101	3 809	5 224	4	5 324	6 784
nestedif2	32	3	31 784	27 952	5	17 269	17 960	197	14 424	15 464	4	17 350	18 176
varops	16	48	2 032	9 680	64	1 361	6 512	64	1 361	6 512	48	2 032	9 680
varops	32	96	4 176	19 920	128	2 801	13 424	128	2 801	13 424	96	4 176	19 920

Table 8.3.: Average Values of the Respective Metrics for all Schemes

	add.lines	QC	TC
Initial Approach (Chap. 6)			
if-stm. w/o additional lines	120.0	286037.4	252612.7
if-stm. w/ additional lines	559.1	129734.5	131327.3
Line-aware Scheme (Sec. 8.1)			
	48.8	558967.7	490699.7
Cost-aware Scheme (Sec. 8.2)			
if-stm. w/o additional lines	120.0	34178.8	67533.0
if-stm. w/ additional lines	559.7	27271.7	58410.7
Cost- & Line-aware Scheme			
	49.5	63610.2	126376.0

is applied. However, both metrics behave complementary. That is, if a designer picks the circuit with the best number of additional lines, he also gets the circuit with the worst circuit costs. This is in line with observations previously made e.g. in [WSMD14].

Nevertheless, combining the line- and cost-aware schemes provides a good trade-off. In doing so, circuits with 49.5 additional lines (just a bit more than the best result) and quantum costs (transistor costs) of 63,610.2 (126,376.0) (less than twice than the best result) are achieved on average.

8.4. Conclusion

To optimize the synthesis of SyReC specifications, we presented two different approaches.

First, we proposed an extended synthesis method, which uses additional gates to uncompute temporary results, allowing former garbage outputs to be re-used as constant inputs. Using this method, each SyReC statement is synthesized without any garbage outputs, but only additional outputs with known constant values. As a consequence, the total number of additional lines can be drastically reduced (e.g. from 653 to 151 for the 32 bit *cpu_alu*). However, this comes at the cost of additional gates (at most doubled gate cost).

The second approach combines common control lines of multiple gates to reduce gate cost. An additional line is needed to temporarily keep the combined value of these control lines. If there is a line with a constant value already present in the respective parts of the circuit, no further line needs to be added. Using this approach, the gate cost can be significantly reduced (e.g. quantum cost from 2,235,491 to 178,783 for the 32 bit *cpu_alu*).

It is up to the designer to decide which metric is most important. Nevertheless, a good compromise between the number of lines and gate cost can be made when combining the two approaches, as shown in the experimental evaluation.

Part III.

Applications

9. Designing a RISC CPU in Reversible Logic

In this chapter, the applicability of a reversible design flow is tested by designing a RISC CPU in reversible logic. Starting from a textual specification, first the core components of the CPU are identified. Previously introduced approaches are applied next to realize the respective combinational and sequential elements. More precisely, the combinational components are designed using the reversible hardware description language SyReC [WOD10], whereas for the realization of the sequential elements an external controller (as suggested in [LP09]) is utilized.

Plugging the respective components together, a CPU design results which can process software programs written in an assembler language. This is demonstrated in a case study, where the execution of a program determining Fibonacci numbers is simulated.

The chapter is structured as follows. The specification of the CPU is provided in Section 9.1, while Section 9.2 discusses the implementation details. Section 9.3 demonstrates the execution of a software program on the proposed CPU. Finally, conclusions are given in Section 9.4.

9.1. Specification of the CPU

In this section, the basic data of the proposed RISC CPU is provided. The specification is inspired by the design of a conventional CPU (see [GKD06]). The CPU was created in order to execute software programs provided in terms of the assembler language shown in Table 9.1. This includes

- 8 arithmetic instructions,
- 8 logic instructions,
- 5 jump instructions, and
- 4 load/store instructions.

The respective assembler programs are transformed into sequences of binary instruction words, which are processed by the CPU. A single instruction word is specified as shown in Figure 9.1 by means of the `ADD` operation. Since in total 25 different instructions are supported, the opcode consists of the five most significant bits of the instruction word (`00111` in case of the `ADD` instruction). The remaining bits give the encoding of the natural numbers i, j, and k, which address the registers used by the instruction.

Table 9.1.: Assembler Instructions for the CPU

Command	Semantic
Arithmetic and Logic Instructions	
`ADC` $R[i]$, $R[j]$, $R[k]$	Addition with carry into $R[i]$
`SBC` $R[i]$, $R[j]$, $R[k]$	Substraction with carry into $R[i]$
`ADD` $R[i]$, $R[j]$, $R[k]$	Addition without carry into $R[i]$
`SUB` $R[i]$, $R[j]$, $R[k]$	Substraction without carry into $R[i]$
`ROR` $R[i]$, $R[j]$	Bitrotation right of $R[j]$
`ROL` $R[i]$, $R[j]$	Bitrotation left of $R[j]$
`SHR` $R[i]$, $R[j]$	Bitshift right of $R[j]$
`SHL` $R[i]$, $R[j]$	Bitshift left of $R[j]$
`NOT` $R[i]$, $R[j]$	Bitwise negation
`XOR` $R[i]$, $R[j]$, $R[k]$	Bitwise exor
`OR` $R[i]$, $R[j]$, $R[k]$	Bitwise or
`AND` $R[i]$, $R[j]$, $R[k]$	Bitwise and
`MKB` $R[i]$, $R[j]$, b	Masking of bit b
`INB` $R[i]$, $R[j]$, b	Inverting of bit b
`SEB` $R[i]$, $R[j]$, b	Set bit b
`CLB` $R[i]$, $R[j]$, b	Clear bit b
Jump Instructions	
`JMP` d	Jump to address d
`JC` d	Jump to address d, if carry is set
`JZ` d	Jump to address d, if zero-flag is set
`JNC` d	Jump to address d, if carry is not set
`JNZ` d	Jump to address d, if zero-flag is not set
Load/Store Instructions	
`LDD` $R[i]$, $R[j]$	Load memory content of address $R[j]$ into $R[i]$
`STO` $R[j]$, $R[k]$	Store $R[k]$ into memory at address $R[j]$
`LDL` $R[i]$, d	Load constant d into low-byte of $R[i]$
`LDH` $R[i]$, d	Load constant d into high-byte of $R[i]$

The CPU has been designed as a Harvard architecture, where the bit width of both, the program memory and the data memory, is 16 bit. The size of the program memory is 4 kByte, while the size of the data memory is 128 kByte. Finally, the CPU has 8 registers, where $R[0]$ always holds the constant 0 and $R[1]$ always holds the constant 1, respectively. All remaining registers are initially assigned to logic 0. As mentioned above, the length of an instruction is 16 bit. Each instruction is executed within one cylce.

Assembler Instruction: `ADD` $R[i],R[j],R[k]$

Instruction format:

15	...	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	bin(i)			-	-	bin(j)		bin(k)	

Figure 9.1.: Instruction Word Representing an ADD Instruction

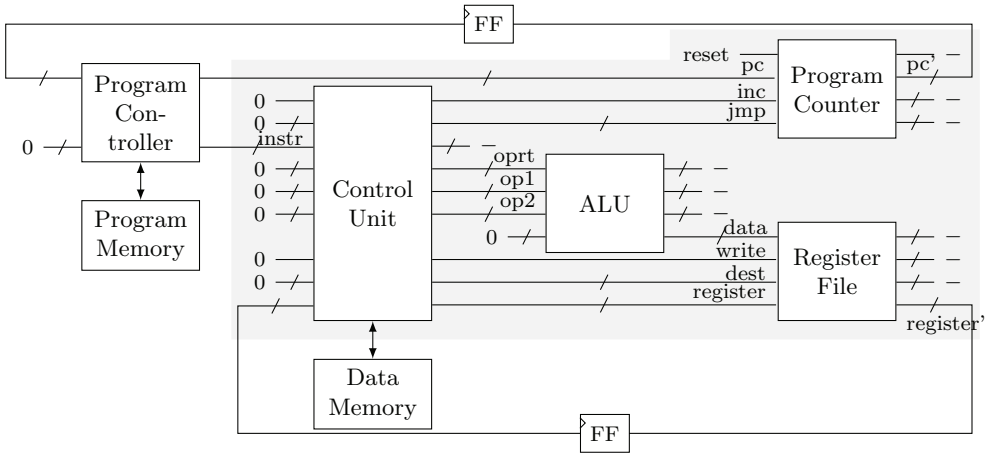


Figure 9.2.: Schematic Diagram of the CPU Implementation

9.2. Implementation of the CPU

The implementation of the above specified CPU is described in this section. Besides an overview, this includes a discussion of the realization of the respective combinational and sequential components. Finally, the characteristics of the resulting circuit are summarized.

9.2.1. Overview

Figure 9.2 provides a schematic overview showing the implementation of the proposed CPU. In the following, the respective components are briefly described from the left-hand side to the right-hand side.

In each cycle, first the current instruction is fetched from the *program memory*. That is, depending on the current value of the program counter `pc`, the respective instruction word is stored in the signal `instr`. Using this signal, the *control unit* decodes the instruction distinguishing between three cases:

1. If an arithmetic or logical operation is performed, the respective operands are extracted from the instruction word and assigned to the signals `op1` and `op2`, respectively. These two signals together with `oprt`, which defines the respective

operation, are passed to the *ALU*. Besides that, the signal `write` is assigned a logic value 1 indicating that the result of the operation should be stored in a target register addressed by `dest`. Finally, the signal `inc` is set to 1, indicating that the program counter has to be increased by 1.

2. If instead a control operation (e.g. a `JMP`) is performed, the signals `op1`, `op2`, `oprt`, `write`, and `dest` are not required for further operation in the current cycle, whereas the signal `inc` is assigned a logic value 0. Further, `jmp` is set to the new address of the program memory depending on the instruction word.

3. A memory access using load and store instructions can be conducted directly by the control unit. In case of an `LDD` instruction, the data is fetched from the memory and stored in the respective register by adjusting the corresponding signal `register`. In contrast, in case of an `STO` instruction, the value of the source register is read and stored in the respective memory address. All other signals are assigned, such that the results of the components are not used (in case of the ALU) or remain unchanged (in case of register file). Also here, signal `inc` is assigned to logic 1.

Afterwards, as defined in the instruction, the respective operation is performed in the *ALU*. Depending on the value of `oprt` as well as the operands `op1` and `op2`, a result is determined and assigned to `data`. This value is then stored in a register addressed by `dest`.

Finally, the program counter is updated. If no control operation has been performed (i.e. if `inc` = 1), the value of signal `pc` is simply increased by one. Otherwise, `pc` is assigned the value given by `jmp`. An exception occurs, if the primary input `reset` is set to 1. Then, the whole execution of the program is reset, i.e. the program counter is set to 0. The updated value of the program counter is used in the next cycle.

Given this CPU architecture, in the following we distinguish between two types of components. Namely:

- Combinational components, i.e. the circuit elements needed to perform the actual computation. This includes the control unit, the ALU, the program counter, and the register file, respectively. That is, all shaded components in Figure 9.2 fall in this category.

- Sequential and memory components, i.e. a clock and flip-flops which are needed e.g. to pass the value of the program counter from one cycle to the next cycle. Also the registers and the memory for both, the program (i.e. the sequence of instructions to be performed) and the data, fall into this category.

In the following, we discuss the state-of-the-art techniques applied in order to realize these components.

9.2.2. Combinational Components

In order to realize combinational reversible circuits, a wide range of synthesis approaches have been introduced in the recent years (see e.g. [MMD03, Ker04, GWDD09, WD09]). Most of them rely on Boolean descriptions such as truth tables or *Binary Decision Diagrams* (BDDs). But since the CPU includes complex operations (e.g. large control paths and arithmetic operations), we used the SyReC programming language as well as its respective synthesis engine to realize the combinational components of the CPU [WOD10].

Thus, the control unit, the ALU, and the program counter can be implemented on a higher level of abstraction. This avoids scalability problems, which would occur if truth-table-based or BDD-based approaches were applied. In contrast, hierarchical synthesis approaches (such as the SyReC engine) tend to generate circuits with a large number of constant inputs. This can partially be improved by post-synthesis optimization approaches (e.g. [WSD10]), but still remains an open problem, which is left for future work. Besides that, new design paradigms have to be considered.

As an example, the SyReC code of the program counter is given in Figure 9.3a. One new design paradigm becomes already evident in this example. According to the specification, the program counter should be assigned 0, if the primary input `reset` is assigned 1. Due to a lack of conventional assignment operations which would destroy the reversibility, this is realized by a new additional signal (denoted by `zero` and set to 0) as well as a SWAP operation (see Line 6 of Figure 9.3a). Similar design decisions have to be made e.g. to realize the desired control path or to implement the respective functionality of the ALU. In contrast, the increase of the program counter is a reversible operation and, thus, can easily be implemented by the respective ++= instruction (Line 9).

The resulting circuit generated by the SyReC synthesizer is shown in Figure 9.3b. Note that the bit widths of the signals are scaled down to 2 in order to improve the readability. The first two lines give the current value of the program counter (`pc_1`, `pc_0`), while the same lines on the right-hand side hold the next state values (`pc_1`$'$, `pc_0`$'$) used as inputs for the flip-flops as depicted in Figure 9.2.

The remaining combinational components are realized similarly. However, due to readability, the complete SyReC code as well as the resulting circuits of all combinational components are not provided in this work. The sources are completely available on RevLib [WGT+08].

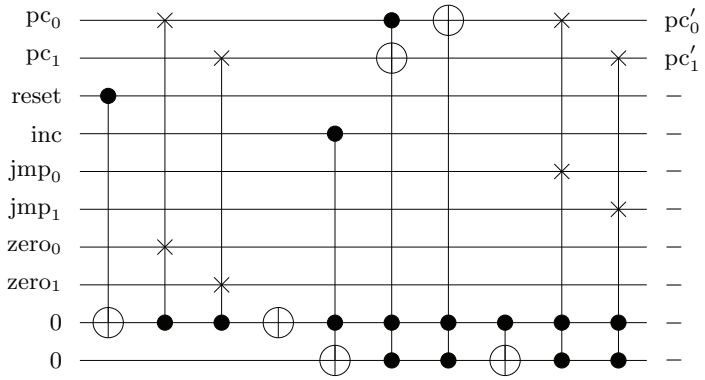
9.2.3. Sequential Components

While for the synthesis of combinational reversible circuits, a significant number of approaches has been introduced, research on design solutions for sequential components is just at the beginning. Two different paradigms are currently under detailed consideration.

The first paradigm (suggested e.g. in [LP09]) arguments that a reversible circuit retains in its state as long as its signal values remain unchanged. Thus, a combinational circuit can be treated as a core component of a sequential device. More precisely, using e.g. a classical (non-reversible) controller, output values from one cycle are applied to

```
1  module pc( inout pc( 2 ), in reset( 1 ), in inc( 1 ), in jmp( 2 ) )
2
3  wire zero ( 2 )
4
5  if ( reset ) then
6      pc <=> zero
7  else
8      if ( inc ) then
9          ++= pc
10     else
11         pc <=> jmp
12     fi ( inc )
13 fi ( reset )
```

(a) SyReC Code



(b) Resulting Circuit

Figure 9.3.: Implementation of the Program Counter (Scaled down to a Bit Width of 2)

the respective input signals of the next cycle. Therefore, the clocking as well as the feedback is handled by the controller, while the actual computation is performed on a combinational reversible circuit.

The second paradigm considers the realization of the sequential elements directly in reversible logic. For this purpose, several suggestions on how to realize the respective memory elements as flip-flops, latches, or registers have been made (see e.g. [TS05, CW07, NHJ+09]). Using these basic sequential elements, more complex sequential components can easily be constructed.

In the actual implementation of the proposed CPU, we decided to realize all sequential components by means of an external controller. Nevertheless, both concepts reviewed above can be applied in principle.

9.2.4. Characteristics of the Resulting Circuit

Using the schematic diagram described in Figure 9.2 and by plugging the synthesized combinational parts together, a reversible circuit results, composed of 1,139 circuit lines (including 867 lines with constant inputs), 5,047 Toffoli gates, and 1,692 Fredkin gates. Considering established cost metrics, this circuit has transistor costs of 504,904 (see [TG08] for more details on transistor costs) and quantum costs of 501,119 (see [BBC+95] for more details on quantum costs)[1]. Together with the external controller for the sequential components, this reversible circuit represents a CPU ready for running programs.

9.3. Executing Programs on the CPU

With the CPU implemented as described in the previous sections, arbitrary software programs composed of the assembler instructions given in Table 9.1 can be executed. Therefore, first an assembler program is translated into a sequence of respective instruction words by applying techniques proposed in [GKD06]. Afterwards, the resulting instruction words are loaded into the program memory, while the data memory is initialized with desired values. Both, the program memory and the data memory, are realized by an external controller implemented in terms of a Python script. Overall, this allows to run translated object code, i.e. a sequence of instruction words.

The execution of a program on the proposed CPU is illustrated using the assembler program depicted in Figure 9.4. Here, the sequence of Fibonacci numbers defined by $f(n) = f(n-1) + f(n-2)$ with $f(0) = f(1) = 1$ is computed. More precisely, the program generates the Fibonacci number $f(n+1)$, whereby $n > 1$ is given in the register $R[7]$. The result is stored in $R[4]$.

The waveform obtained by simulating this program (with $n = 4$) on the CPU is given in Figure 9.5. The identifiers *clk*, *pc'*, and *instr[15:11]* denote the values of the clock signal, the program counter, and the operation code extracted from the `instr` signal,

[1] Note that these costs probably can be significantly reduced by applying technology depend post-synthesis approaches.

```
0  LDL R[7], 4
1  LDL R[2], 1
2  LDL R[3], 1
   loop:
3  ADD R[4], R[3], R[2]
4  OR  R[2], R[3], R[0]
5  OR  R[3], R[4], R[0]
6  SUB R[7], R[7], R[1]
7  JNZ loop
```

Figure 9.4.: Assembler Program for Fibonacci Number Computation

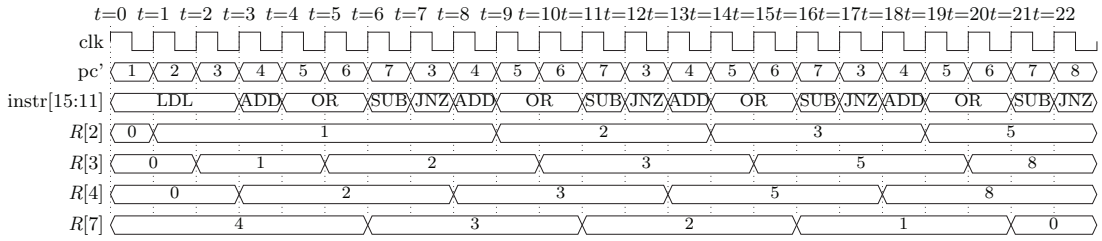


Figure 9.5.: Waveform Illustrating the Execution of the Program Given in Figure 9.4

respectively. The rows $R[2]$, $R[3]$, $R[4]$, and $R[7]$ list the values of the respective registers. For the sake of clarity, all other signal values are omitted. Note that the value of the program counter always corresponds to the respective line number of the code given in Figure 9.4. In each time frame always the updated values of the signals obtained after the execution are listed.

At the beginning of the execution, the registers are loaded with the given values, i.e. $R[7]$ is assigned 4, while $R[2]$ and $R[3]$ are assigned the first two Fibonacci numbers, respectively ($t = 0$ until $t = 2$). Next, the third Fibonacci number is determined by adding the values of $R[3]$ and $R[2]$. The result is assigned to $R[4]$ ($t = 3$). The following `OR` operations update the auxiliary values of the registers $R[2]$ and $R[3]$ ($t = 4$ and $t = 5$). Recall that according to the specification provided in Section 9.1, the register $R[0]$ always holds the constant 0, i.e. register $R[2]$ is assigned the value of $R[3]$, while the register $R[3]$ is assigned the value of $R[4]$. Now the values for the next iteration are available. But before starting the next iteration, the loop bound stored in $R[7]$ needs to be decreased by one. For this task, the register $R[1]$ – which always holds the constant 1 – is used. Afterwards, the jump instruction is processed modifying the program counter so that the previous steps are repeated with the current values ($t = 7$). This execution continues as long as the value in register $R[7]$ is not 0. Finally, the result of the computation can be obtained from the value assigned to $R[4]$. For the given example program we get $f(4 + 1) = f(5) = 8$.

9.4. Conclusion

In this chapter, we proposed a design of a RISC CPU realized using reversible logic. Therefore, recent achievements in the domain of reversible circuit design have been employed. In particular, this includes the hardware description language SyReC, which has been used to design the combinational components of the CPU. In contrast, the sequential components have been realized using an external controller. With the proposed CPU, it is possible to execute software programs using an assembler language. Besides that, the circuit can be used as benchmark for other areas such as the verification or the test of reversible circuits. Therefore, the CPU has been made public available at RevLib [WGT+08].

Future work is focused on the optimization of the resulting circuit. As discussed in Section 9.2.2, in particular reducing the number of lines is important. For this purpose, one could consider the approach presented in [WSD10]. Furthermore, having the CPU design available, a physical realization of a complex application is possible. So far, only simple circuits have been physically realized. Finally, the design of a CPU processing reversible software languages (as e.g. Janus [YG07]) may provide an interesting case study.

10. Visualization of Structures and Properties of Reversible Circuits

For the design, synthesis, debugging etc. of reversible circuits, a lot of ideas have been taken from the conventional design flow. But besides the apparent way of proceeding, human intuition often led to ideas for new strategies to be exploited or enabled further improvements which could not be detected by a machine.

However, getting a good intuition of a considered circuit requires a deep technical understanding of how design approaches actually realize the respective circuits. Moreover, these approaches may generate circuits with certain structures and properties that are often neither obvious to the developer nor to the user of the design method. Consequently, possible potential in terms of better synthesis or optimization may often not fully be exploited.

In fact, relevant instances of any kind are often equipped with some internal (sometimes hidden) structures or properties that are unknown to the developer and/or designer [Wal99]. One way to unveil these information is to provide a different intuition about a circuit. This can be accomplished by visualization technologies. However, existing visualization schemes for reversible circuits are basically limited to simple netlist representations in which all gates are only arranged in a cascade where black circles and $\oplus$ respectively represent control and target lines of the gates. In particular for larger circuits, these netlists do not provide a proper intuition of the structure and possible properties of reversible circuits. As an example, consider the netlist visualization of a circuit realizing a division and shown in Fig. 10.1 (realized by the HDL-based synthesis approach proposed in Chapter 6). Although this circuit is composed of less than 100 gates, it is almost impossible to recognize certain structures and/or properties from this netlist visualization.

As a consequence, advanced visualization techniques are required that go beyond the straight-forward representation of a circuit as a netlist. They should mask irrelevant details as deemed necessary and, in turn, explicitly focus on highlighting the desired structures and properties. In other domains, such visualization techniques have already

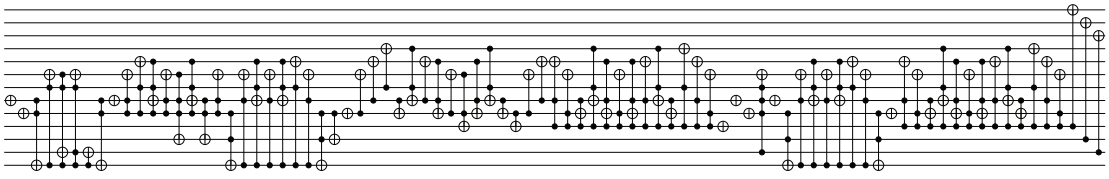
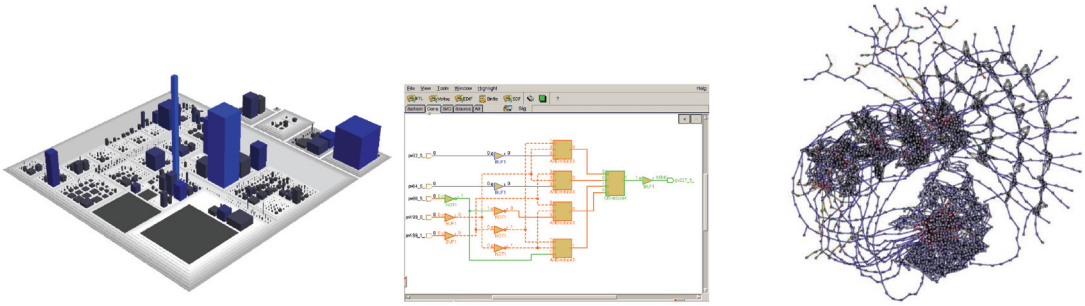


Figure 10.1.: Existing Netlist Visualization of Reversible Circuits



(a) SW Design (CodeCity) (b) HW Debugging (c) SAT Solving

Figure 10.2.: Visualization Technologies in Other Domains

successfully been applied. For example:

- In the conventional software design, visualizations such as the *CodeCity* [WLR11] are well known. Here, different software classes are placed as "buildings" within an artificial representation of a city. Depending on their properties, e.g. their number of attributes, methods, or lines of code, the ground size or the height of the "buildings" differ. Structural interrelation between classes is e.g. emphasized by placing the corresponding "buildings" in the same "district". Fig. 10.2a shows such a visualization taken from [WLR11]. Unproportional looking "buildings" immediately pinpoint the designer to problematic classes in the software project. The visualization reveals classes that are too complex in terms of code and may better be split into subclasses or are not well-balanced in terms of their number of attributes to number of methods ratio.

- In the domain of debugging (conventional) hardware, so called *error candidates* are explicitly highlighted in the netlist [SWG+09]. They represent logic elements within the circuit that may explain an erroneous behavior. Fig. 10.2b shows such a visualization (taken from [SWG+09]). By this, the designer is explicitly pinpointed to possible reasons for the incorrect behavior and does not have to consider all gates of the circuit at once. Furthermore, by lapping several of such layers, the designer is provided with an intuitive representation of the circuit as well as possible explanations for the error which aids him/her during the debugging process.

- Solvers for Boolean satisfiability (so called SAT solvers [ES04]) have been shown to be very powerful and, hence, find practical applications e.g. in domains like verification. However, although these approaches are able to efficiently solve instances composed of hundreds of thousands of variables and constraints, much smaller instances remain unsolvable within generous time limits. Understanding what makes a SAT instance hard or not has also been investigated using visualization technologies [Sin07]. For this purpose, instances have been represented by graphs as shown in Fig. 10.2c (taken from [Sin07]), where nodes represent the variables of the instance and edges the constraints over them. Using a visualization

like this intuitively unveils connected sub-functions, important and less important literals, etc. This provides a better understanding about how instances could be solved in a more efficient fashion.

Motivated by these success stories, the application of visualization technologies in the domain of reversible circuit design is investigated in this work. For this purpose, we present the tool *RevVis*, a graphical interface that intuitively visualizes the structure and properties of reversible circuits. For a selected set of metrics and objectives which are relevant in the design of reversible circuits, corresponding data is collected and, afterwards, visualized in a simple fashion. The application of *RevVis* has been evaluated in a thorough case study involving several synthesis approaches that have been proposed in the past. From the different visualizations some already known structures and properties could be confirmed. Beyond that also new characteristics could be unveiled. They may be exploited in the future to further finetune these approaches and to develop corresponding new optimization schemes for the resulting circuits.

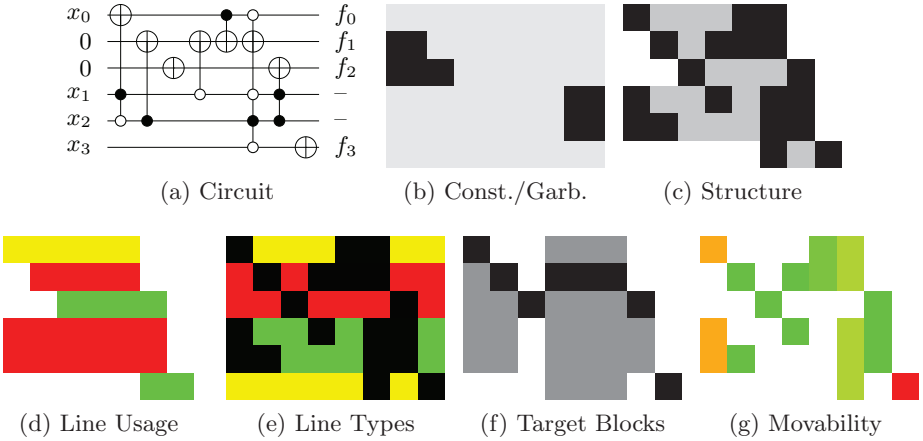
The remainder of this chapter is structured as follows. Section 10.1 introduces *RevVis* and, in particular, the visualizations of the selected metrics and objectives. Afterwards, these visualizations are applied for circuits generated by several synthesis approaches. Possible conclusions drawn from that are discussed in Section 10.2. The chapter is eventually concluded in Section 10.3.

10.1. The RevVis Tool

This section introduces the main features of the proposed visualization schemes which have been implemented in the tool *RevVis*[1]. For a selected set of metrics and objectives, the tool first collects information on the structure and properties of a given reversible circuit, which are then visualized. The visualizations are kept as simple and abstract as possible so that, even for larger designs, an intuitive and easy understanding is possible. In the following, the considered metrics and objectives are introduced. Here, all visualization schemes are illustrated by means of the reversible circuit depicted in Fig. 10.3a.

Constant Inputs and Garbage Outputs. Constant inputs and garbage outputs are not only essential in order to embed irreversible functions into reversible ones (see e.g. [MD04a, WKD11]), but are also heavily applied in synthesis approaches e.g. based on ESOPs (e.g. [FTR07]) or decision diagrams (e.g. [WD09]). Optimization approaches such as introduced in [WSD10] rely on the fact how long circuit lines with constants or garbage are unused or not needed anymore, respectively. This is emphasized by the first visualization scheme shown in Fig. 10.3b. All lines inheriting a constant or garbage line are highlighted by black rows. The width of the rows depends on the number of gates in the cascade in which the respective constant (garbage) is unused (not needed anymore).

[1] RevVis is available at http://www.informatik.uni-bremen.de/agra/eng/revvis.php.



(a) Circuit (b) Const./Garb. (c) Structure

(d) Line Usage (e) Line Types (f) Target Blocks (g) Movability

Figure 10.3.: Different Visualizations in RevVis

Structure of the Circuit. Reversible circuits are composed as a cascade of reversible gates which, in turn, are composed of control lines and target lines. Due to this cascade structure, the structural usage of each line in a circuit may significantly differ. This is visualized in the scheme shown in Fig. 10.3c. Each control and target line connection is highlighted in black. Grey denotes the usage of each circuit line, i.e. the cascade from the first gate in which this circuit line is involved until the last gate of the cascade. White represents parts of the circuit which are not needed for the actual computation. For example, the bottom line of the considered circuit is only needed at the end of the cascade while all remaining lines are needed almost throughout the whole cascade. Although similar to the netlist visualization, this simplified view enables a more intuitive view on the structure of a circuit and can pinpoint to "holes" in the circuit (which can be used e.g. as ancilliae).

Line Usage. The usage of circuit lines is additionally visualized by the scheme shown in Fig. 10.3d. Here, the visualization is enriched by a color code representing the numerical usage of a circuit line. Lines highlighted red (green) represent the circuit lines with the largest (smallest) number of control and target line connections. Yellow patterns denote the circuit lines which lie between these extremes. White represents parts of the circuit which are not needed for the actual computation. Information like that could e.g. be applied for nearest neighbor optimization (see e.g. [SWD11, AAAH13, SSP13, WLD14]). Here, control and target line connections always have to be adjacent, i.e. lines which are heavily used should preferably be put next to each other.

Line Types. The distribution of control and target line connections is an objective of the scheme shown in Fig. 10.3e. Here, red lines (green lines) denote circuit lines which are entirely composed of target lines (control lines) only; yellow lines denote circuit lines which have both control and target line connections. All actual connections are again

highlighted in black. This could provide some inspiration for optimization as e.g. huge parts of the circuit composed entirely of control lines may provide some potential for reduction by factorization (see e.g. [MWD10]).

Target Blocks. Fig. 10.3f shows another scheme which focuses on the target line connections. More precisely, sub-circuits in which all gates have the same target line are highlighted by means of grey blocks (with the target lines additionally highlighted in black). Also this view could provide some inspiration for optimization (in particular, if the possibly different control connections could be merged so that such a cascade can be reduced to some few or even a single gate(s)).

Movability. Finally, the "movability" of gates is visualized in Fig. 10.3g, i.e. the applicability of the moving rule as reviewed in Chapter 2 is represented for each gate. Gates highlighted red have a low movability (i.e. can hardly be moved through the cascade), while gates highlighted green can be moved rather flexibly through the cascade. Obviously this view is particularly helpful to investigate optimization approaches relying on the moving rule.

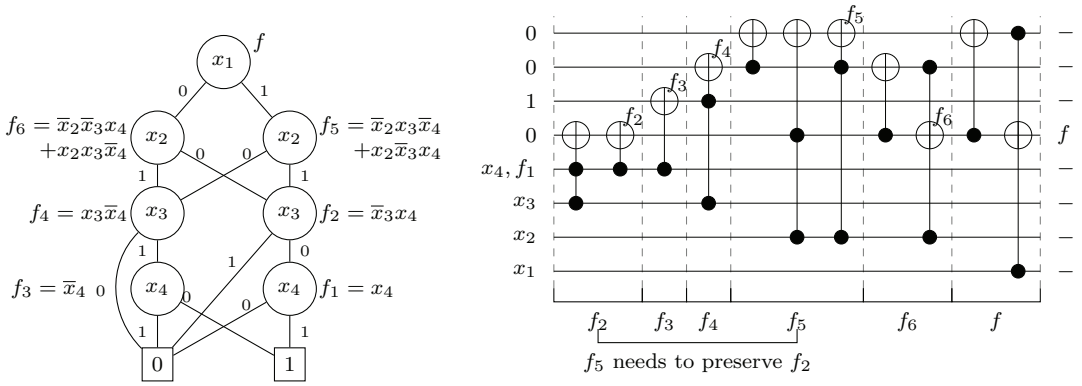
10.2. Applying RevVis

The visualizations proposed in the last section are supposed to provide a representation which allows to grasp a good intuition of the structure and the properties of a given circuit. In order to illustrate that *RevVis* satisfies this purpose, an intense case study has been conducted, in which circuits generated with different synthesis approaches (namely BDD-based synthesis [WD09], ESOP-based synthesis [FTR07], and HDL-based synthesis [WOD10, WSSD12]) have been investigated using *RevVis*. In this section, results of these investigations are exemplarily shown and discussed. For this purpose, first the respective synthesis approach is briefly reviewed. Afterwards, a representative circuit (taken from RevLib [WGT+08]) is visualized and corresponding observations are discussed.

10.2.1. Considering Circuits Obtained by BDD-based Synthesis

The Synthesis Approach (see also Chapter 3)

BDD-based synthesis as introduced in [WD09] makes use of *Binary Decision Diagrams* (BDDs) [Bry86]. A BDD is a directed graph $G = (V, E)$ where each terminal node represents the constant 0 or 1 and each non-terminal node represents a (sub-)function. Each non-terminal node $v \in V$ has two succeeding nodes low(v) and high(v). If v is representing the function f and labeled with the variable x_i, then the corresponding sub-functions represented by the succeeding nodes are the co-factors $f_{x_i=0}$ (low(v)) and $f_{x_i=1}$ (high(v)). Thus, a BDD naturally exposes the Shannon decomposition. Having a BDD representing a function f as well as its sub-functions derived by Shannon decomposition,



(a) BDD (b) Resulting Circuit

Figure 10.4.: BDD-based Synthesis

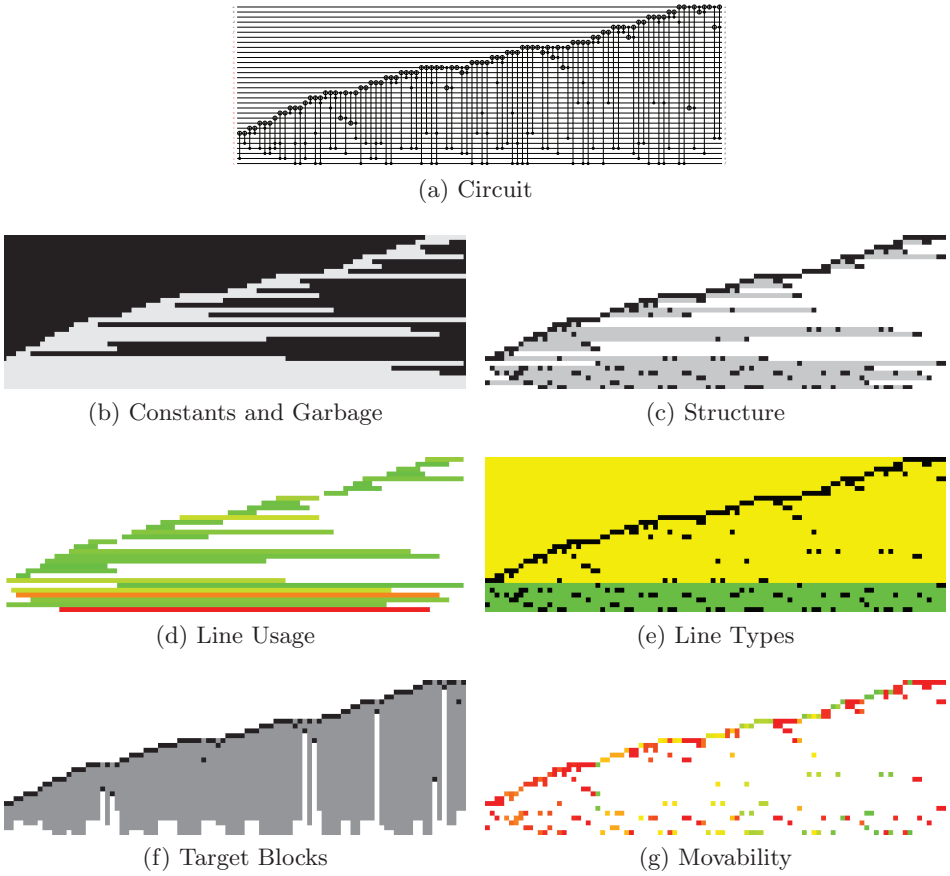
a reversible circuit for f can be obtained as shown by the following example (taken from Chapter 3).

Example 15 *Fig. 10.4a shows a BDD representing the function* $f = \overline{x}_1\overline{x}_2\overline{x}_3x_4 + \overline{x}_1x_2x_3\overline{x}_4 + x_1\overline{x}_2x_3\overline{x}_4 + x_1x_2\overline{x}_3x_4$ *as well as the respective co-factors resulting from the application of the Shannon decomposition. The co-factor* f_1 *can easily be represented by the primary input* x_4*. Having the value of* f_1 *available, the co-factor* f_2 *can be realized by the first two gates depicted in Fig. 10.4b*[2]*. By this, respective sub-circuits can be added for all remaining co-factors until a circuit representing the overall function* f *results. The remaining steps are shown in Fig. 10.4b.*

Observations Using RevVis

Fig. 10.5 shows the visualizations for the circuit *mod5adder_66* which has been obtained using BDD-based synthesis and works as a proper representative for this synthesis scheme. Compared to the simple netlist (see Fig. 10.5a), these visualizations unveil the clear structure of these circuits. In fact, BDD-based synthesis heavily relies on constant inputs (see Fig. 10.5b) and subsequently builds up the sub-functions (i.e. the co-factors) of the BDD. This can clearly be seen in Figs. 10.5c and 10.5f: New functionality is costantly build up towards the top-right of the circuit. The primary inputs (located at the bottom of the circuit) are frequently used for this purpose. This explains the intense usage of these circuit lines (see Fig. 10.5d). It also shows very nicely that the usage of the primary inputs depends on the BDD-level, e.g. the primary input represented by the root node of the BDD has a very low usage while primary inputs represented in lower levels of the BDD are accessed more often. As shown in Fig. 10.5e, all primary input lines are accessed in a read-only fashion (i.e. just control connections are applied in those

[2]Note that an additional circuit line is added to preserve the values of x_4 and x_3 which are still needed by the co-factors f_3 and f_4, respectively.



(a) Circuit

(b) Constants and Garbage

(c) Structure

(d) Line Usage

(e) Line Types

(f) Target Blocks

(g) Movability

Figure 10.5.: Visualizing a Circuit Obtained by BDD-based Synthesis

circuit lines). Finally, Fig. 10.5g unveils that movability is usually rather bad in circuits generated by BDD-based synthesis.

By this, several properties of BDD-based circuits which are already known (e.g. the huge number of constant/garbage) are confirmed. Besides that, a clearer intuition of the actual structure and properties is provided. For example, Fig. 10.5b may offer more precise hints where to merge constants and garbage (similar to the approach presented in [WSD10]). Fig. 10.5g clearly shows that e.g. optimization approaches like template matching [MMD03] (relying on the moving rule) are not really suitable for BDD-based circuits. Besides that, the clear stepped structure of the overall circuit might be exploitable for further optimizations.

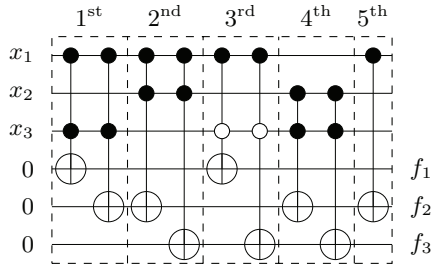
10.2.2. Considering Circuits Obtained by ESOP-based Synthesis

The Synthesis Approach

ESOP-based synthesis as introduced in [FTR07] generates a reversible circuit from a Boolean function provided as *Exclusive Sum of Products* (ESOPs). ESOPs are two-

	x_1	x_2	x_3	f_1	f_2	f_3
1^{st}	1	-	1	1	1	0
2^{nd}	1	1	-	0	1	1
3^{rd}	1	-	0	1	0	1
4^{th}	-	1	1	0	1	1
5^{th}	1	-	-	0	1	0

(a) ESOP



(b) Resulting Circuit

Figure 10.6.: ESOP-based Synthesis

level descriptions of Boolean functions that are represented as the exclusive disjunction (EXOR) of conjunctions of literals (called *products*). A *literal* is either a Boolean variable or its negation. That is, an ESOP is the most general form of two-level AND-EXOR expressions.

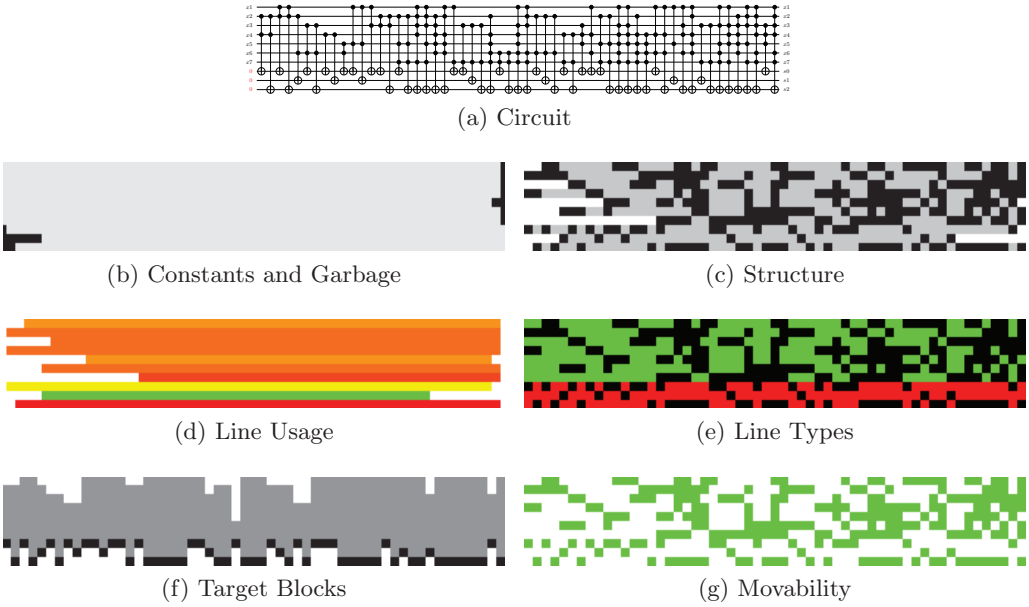
Having an ESOP representing a function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$, the ESOP-based synthesis approach generates a circuit with $n + m$ lines, where the first n lines work as primary inputs, while the last m circuit lines are initialized to constant 0 and work as primary outputs. Having that, Toffoli gates are selected such that the desired function is realized. This selection exploits the fact that a single product $x_{i_1} \dots x_{i_k}$ of an ESOP description directly corresponds to a Toffoli gate with control lines $C = \{x_{i_1}, \dots, x_{i_k}\}$. In case of negative literals, NOT gates or negative control lines are applied accordingly. Based on these ideas, a circuit realizing a function given as ESOP can be derived as illustrated in the following example.

Example 16 *Consider the function f to be synthesized as depicted in Fig. 10.6a[3]. The first product x_1x_3 affects f_1 and f_2. Hence, two Toffoli gates which have target lines f_1 and f_2 and control lines $C = \{x_1, x_3\}$ are added (see Fig. 10.6b). The third product $x_1\overline{x}_3$ includes a negative literal. Thus, the Toffoli gates added for this product have a negative control line on x_3. This procedure is continued until all products have been considered. The resulting circuit is shown in Fig. 10.6b.*

Observations Using RevVis

Fig. 10.7 shows the visualizations for the circuit *rd73_252* which has been obtained using ESOP-based synthesis and works as a proper representative for this synthesis scheme. Compared to the simple netlist (see Fig. 10.7a), the characteristic structure is clearly unveiled thanks to the visualizations. In particular, the distinction between

[3]The column on the left-hand side gives the products, where a "1" on the i^{th} position denotes a positive literal (i.e. x_i) and a "0" denotes a negative literal (i.e. $\overline{x}_i$), respectively. A "–" denotes that the respective variable is not included in the product. The right-hand side gives the primary output patterns.



(a) Circuit

(b) Constants and Garbage

(c) Structure

(d) Line Usage

(e) Line Types

(f) Target Blocks

(g) Movability

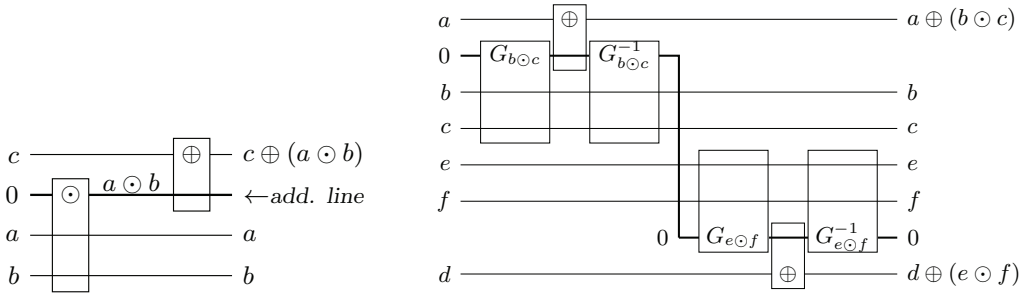
Figure 10.7.: Visualizing a Circuit Obtained by ESOP-based Synthesis

input lines (which have control connections only) and output lines (which have target connections only) becomes evident (see Fig. 10.7e) and also leads to a very regular structure with respect to target blocks (see e.g. Fig. 10.7f). This provides potential as it may allow to merge gates with equal control lines but different target lines (as discussed e.g. in [WSOD13]). Furthermore, approaches relying on the moving rule (e.g. [MMD03]) significantly benefit from this structure as it leads to a very high movability (see Fig. 10.7g). It may also be observed that, due to the high movability of gates, many target blocks can be merged leading to more potential for optimization. In contrast, constant inputs are used very early in the cascade (see Fig. 10.7b), i.e. there is no potential to reduce the number of constant/garbage lines using e.g. the method proposed in [WSD10]. Besides that, ESOP-based circuits seem to have a rather irregular structure, i.e. the respective gate connections are distributed rather arbitrarily (see Fig. 10.7c). However, it can be observed that inputs lines are used more often than output lines (see Fig. 10.7d). This can be explained by the fact that some factors may have to be applied to several functions and, hence, identical control connections are frequently applied.

10.2.3. Considering Circuits Obtained by HDL-based Synthesis

The Synthesis Approach (see also Chapter 5 to 8)

The strive for more scalable synthesis approaches also led to the definition and consideration of a *Hardware Description Language* (HDL) for reversible circuits in [WOD10]. In order to ensure reversibility in the description, this HDL distinguishes between reversible assignments (denoted by $\oplus=$) and not necessarily reversible *binary operations*



(a) Straight-forward (b) Improved

Figure 10.8.: HDL-based Synthesis

(denoted by $\odot$). The former class of operations assigns values to a signal on the left-hand side. Therefore, the left-hand side signal must not appear in the expression on the right-hand side. Furthermore, only a restricted set of assignment operations exists, namely increase (`+=`), decrease (`-=`), and bitwise XOR (`^=`). These operations preserve the reversibility (i.e. it is possible to compute these operations in both directions). In contrast, binary operations, e.g. arithmetic, bitwise, logical, or relational operations, may not be reversible and, hence, can only be used in right-hand expressions which preserve the values of the inputs. In doing so, all computations remain reversible since the input values can be applied to reverse any operation. For example, to describe a multiplication (i.e. `a*b`), a new free signal `c` must be introduced which is used to store the product (i.e. `c^=a*b` is applied). In comparison to common (non-reversible) languages, this forbids statements like `a=a*b`.

Having such an HDL description, synthesis approaches like introduced in [WOD10] generate corresponding circuits following a hierarchical scheme. That is, existing realizations of the individual operations (i.e. building blocks) are combined so that the desired circuit is realized. This is illustrated in Fig. 10.8a for the generic operation $c\oplus = (a \odot b)$. First, the binary operation $\odot$ is realized (using additional circuit lines with constant inputs). Afterwards, the intermediate result is utilized to realize the complete statement including its reversible assignment $\oplus=$.

This scheme has further been improved in [WSSD12]. Here, the values of intermediate results are reversed once they are not needed any longer (leading back to the original constant value). Then, no new additional lines might be required to buffer upcoming intermediate results. The general idea is briefly illustrated in Fig. 10.8b by means of the generic HDL statements $a\oplus = (b \odot c)$ and $d\oplus = (e \odot f)$. First, two sub-circuits $G_{b\odot c}$ and $G_{a\oplus=b\odot c}$ are added ensuring that the first statement is realized. This is equal to the procedure from Fig. 10.8a and leads to additional lines with constant inputs. But then, a further sub-circuit $G^{-1}_{b\odot c}$ is applied. Since $G^{-1}_{b\odot c}$ is the inverse of $G_{b\odot c}$, this sets the circuit lines buffering the result of $b \odot c$ back to the constant 0. As a result, these circuit lines can be reused in order to realize the following statements as illustrated for $d\oplus=e \odot f$ in Fig. 10.8b.

Observations Using RevVis

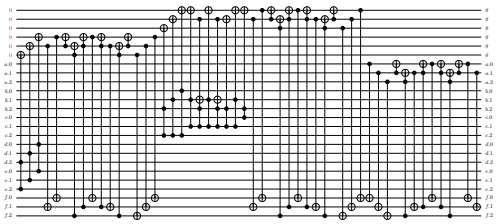
Fig. 10.9 (Fig. 10.10) shows the visualizations for the circuit *mult_stmts_3bit* which has been obtained using the straight-forward HDL-based synthesis as illustrated in Fig. 10.8a (the improved HDL-based synthesis as illustrated in Fig. 10.8b) and works as a proper representative for this synthesis scheme. More precisely, these circuits realize three HDL-statements over 3-bit variables. The respective cascades for each statement are separated by vertical lines in Fig. 10.9 and Fig. 10.10. Compared to the simple netlist (see Fig. 10.9a and Fig. 10.10a), these visualizations do not only unveil the structure and characteristics of the respective circuits, but also the differences between the straight-forward and optimized synthesis scheme.

First of all, the structures sketched in Fig. 10.8, i.e. the building blocks for binary operations, reversible assignments, and reversing, can also be recognized in the visualizations (see e.g. Fig. 10.9c and Fig. 10.10c). In particular for the improved scheme, the symmetry resulting from reversing intermediate results is rather obvious. Here, it can also be observed that just one set of constant circuit lines is needed, while the straight-forward approach uses several constant circuit lines only for a short time (compare Fig. 10.9b and Fig. 10.10b). The frequent re-use of these lines in the improved approach is also reflected in the line usage visualization (see Fig. 10.10d).

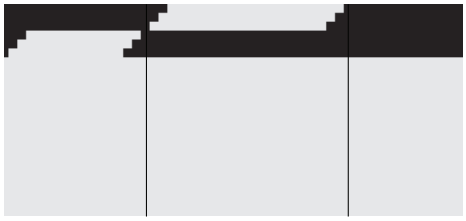
Besides that, many circuit lines only have control connections in this example (see Figs. 10.9e and 10.10e). This is caused by the fact that the three HDL-statements are of the form $a \oplus = (b \odot c)$, i.e. b and c never occur on the left-hand side of a statement. Finally, the visualization clearly unveils that HDL-based circuits have a rather poor movability and, hence, do not seem very suitable for optimization schemes such as [MMD03] (see Figs. 10.9g and 10.10g).

10.3. Conclusion

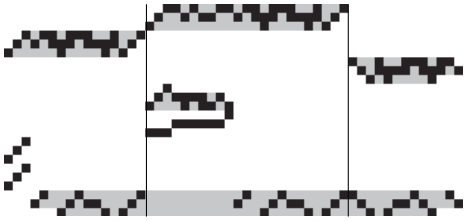
In this chapter, we considered the visualization of reversible circuits. This is motivated by the fact that certain structures and properties of circuits are often not obvious to the developer or to the user. Furthermore, simple netlist representations do not provide a proper intuition and, hence, are not suitable – particularly for circuits of larger size. In order to address this, we introduced the tool *RevVis* which provides visualization layers for several metrics as well as objectives and, by this, intuitively highlights structures and properties of reversible circuits. The application of *RevVis* has been evaluated in a thorough case study involving several synthesis approaches. This enabled a deeper discussion about both known as well as new characteristics of the obtained circuits and, hence, the considered synthesis schemes. In the future, visualizations as proposed in this work will be beneficial to draw conclusions from newly developed design approaches right from the beginning as well as to gain inspiration for new synthesis and optimization methods.



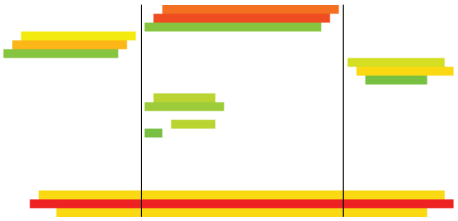
(a) Circuit



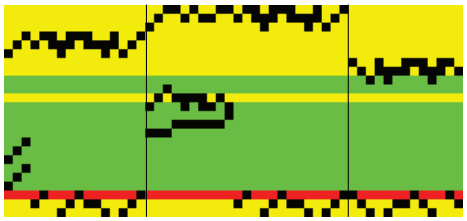
(b) Constants and Garbage



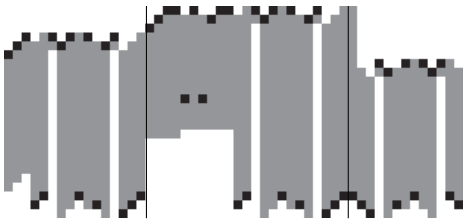
(c) Structure



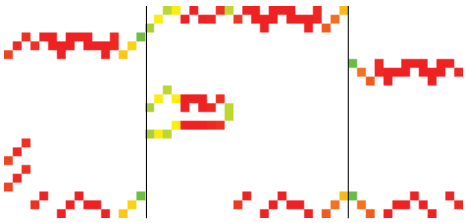
(d) Line Usage



(e) Line Types

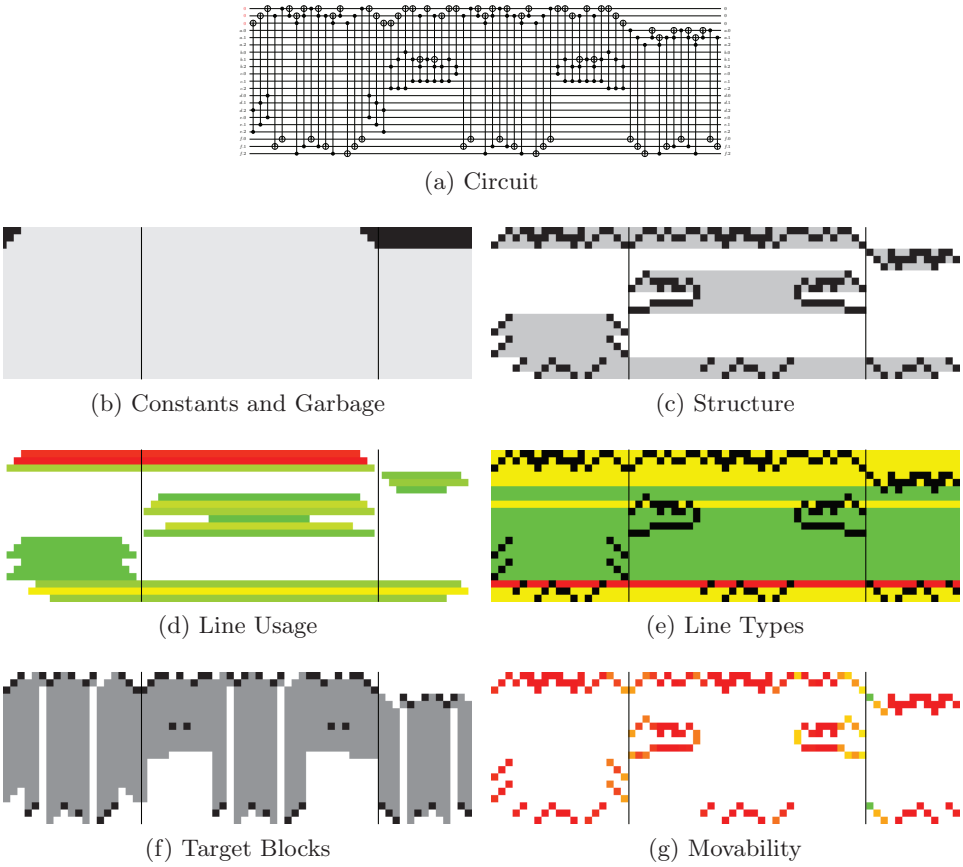


(f) Target Blocks



(g) Movability

Figure 10.9.: Visualizing a Circuit Obtained by HDL-based Synthesis



(a) Circuit

(b) Constants and Garbage

(c) Structure

(d) Line Usage

(e) Line Types

(f) Target Blocks

(g) Movability

Figure 10.10.: Visualizing a Circuit Obtained by Improved HDL-based Synthesis

11. Conclusion

In the last decades, computer technology has advanced extremely fast, and the demands are constantly increasing. Conventional technologies will sooner or later reach their limits regarding miniaturization, energy consumption, etc. Reversible computation finds application in many promising alternatives such as quantum computation or adiabatic circuits. While there has been a lot of research on reversible logic especially in the last decade, most of it considered the design and synthesis of circuits on a rather small scale. There has been very few work on the design and implementation of complex systems in reversible logic. To actually create an alternative to conventional CMOS technology, a design flow for reversible circuits has to be developed.

In this thesis, we investigated two different approaches to designing and synthesizing complex systems in reversible logic.

The first approach is to exploit the conventional design flow, and eventually map a regular circuit design to a reversible one. Here we optimized a synthesis algorithm which creates reversible circuits from decision diagrams, a data structure used in the conventional design flow. The negative controls that were introduced to the method match especially well with the function representation of KFDDs, so the number of gates could be significantly reduced. Furthermore, we created a direct mapping from the register transfer level of a conventional circuit to a reversible circuit. A first evaluation indicates that this algorithm gives results similar to the SyReC synthesis [WOD10] and might be a promising alternative.

With this approach, the existing elaborated design flow including languages, data structures and tools can be utilized to develop reversible logic. For example, a design engineer could write a Verilog or VHDL description which is automatically translated to the register transfer level (using an existing tool) and then automatically translated to a reversible circuit (using our proposed mapping). This would include the advantage of design engineers not having to adapt to the new technology. Additionally, future developments in conventional design could directly be exploited.

The drawback of this approach is that reversibility is not considered throughout the design process. Mappings from conventional designs to reversible circuits usually include embedding of non-reversible functionality. This and the hierarchical approach in general make it hard to find efficient mappings. The resulting circuits will be of high costs and have to be optimized, which is also a difficult task.

The second approach we considered is to develop a specific design flow for reversible logic. We revised SyReC, an HDL for reversible circuits, and its synthesis algorithm. While it enables the design and synthesis of complex systems in reversible logic, the

circuits generated usually have a high amount of additional circuit signals and gate cost. Therefore, we proposed an extended synthesis algorithm, which can save a significant number of additional signals at the cost of generating more gates. Depending on the application, this can be a lucrative trade-off.

In this approach, reversibility is considered at all stages of the design process, so theoretically there is no need for embedding. Synthesis methods can exploit the reversibility to directly obtain efficient realizations.

On the other hand, the design and synthesis methods need to be developed first, especially at higher abstraction levels. Also, languages for all abstraction levels need to be defined. The new computing paradigm would require the training of design engineers.

We tested the specific design flow by designing a RISC CPU in reversible logic. In this case study, we could efficiently represent the computational components of the CPU in SyReC, and thus generate the respective reversible netlists. The meaning of "new computation paradigm" came to light when we needed to handle direct assignments in the functionality and memory components in the CPU.

Last but not least, we created a tool to visualize structures and properties of reversible circuits. To get a first intuition, we investigated circuits created by different synthesis methods. These kind of visualizations can surely be an inspiration for new synthesis methods and optimizations.

With this work, we present possible solutions for the scalable design and synthesis of reversible ciruits. Although the preliminary evaluation of them just provides a rough estimate of the resulting quality, it clearly shows that both directions have potential. To fully exploit this potential, the challenges discussed in the paragraphs (and corresponding chapters) above have to be addressed.

Furthermore, an interesting task is to create a mapping from a conventional HDL to a HDL for reversible computation. Accordingly, it should be investigated further which step in the conventional design flow would be the best step to map to a reversible description, and if it should be mapped to the same level (like HDL to HDL) or not (like RTL to netlist). The SyReC synthesis would benefit a lot from creating an intermediate language (between HDL and netlist) and the use of compiler optimization techniques. To compare both design flow directions, more case studies or benchmarks in the dimension of the RISC CPU are needed. Thus far, the design flow for reversible circuits is far behind the conventional one. The solutions proposed in this thesis are the first step to close this gap.

Bibliography

[AAAH13] M. Alfailakawi, L. Alterkawi, I. Ahmad, and S. Hamdan. Line ordering of reversible circuits for linear nearest neighbor realization. *Quantum Information Processing*, 12(10):3319–3339, October 2013.

[Axe11] H. B. Axelsen. Clean translation of an imperative reversible programming language. In *Int'l Conf. on Compiler Construction*, pages 144–163, 2011.

[BAP+12] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483:187–189, 2012.

[BBC+95] A. Barenco, C. H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary Gates for Quantum Computation. *The American Physical Society*, 52:3457–3467, 1995.

[BDW95] B. Becker, R. Drechsler, and R. Werchner. On the Relation Between BDDs and FDDs. In *Information and Computation*, pages 72–83, 1995.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[CW07] M. Chuang and C. Wang. Synthesis of Reversible Sequential Elements. In *ASP Design Automation Conf.*, pages 420–425, 2007.

[DB06] R. Drechsler and B. Becker. Ordered Kronecker Functional Decision Diagrams-a Data Structure for Representation and Manipulation of Boolean Functions. *IEEE Trans. on CAD*, 17(10):965–973, 2006.

[DRW+13] K. Datta, G. Rathi, R. Wille, I. Sengupta, H. Rahaman, and R. Drechsler. Exploiting Negative Control Lines in the Optimization of Reversible Circuits. In *Int'l Conf. on Reversible Computation*, pages 209–220, 2013.

[DSR13] K. Datta, I. Sengupta, and H. Rahaman. Particle Swarm Optimization Based Reversible Circuit Synthesis Using Mixed Control Toffoli Gates. *Journal of Low Power Electronics*, 9(3):363–372, 2013.

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams. In *Design Automation Conf.*, pages 415–419, 1994.

[DV02] B. Desoete and A. D. Vos. A reversible carry-look-ahead adder using control gates. *INTEGRATION, the VLSI Jour.*, 33(1-2):89–104, 2002.

[ES04] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004.

[Fey85] R. Feynman. Quantum mechanical computers. *Optic News*, 11:11–20, 1985.

[FT82] E. F. Fredkin and T. Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, 21(3/4):219–253, 1982.

[FTR07] K. Fazel, M. A. Thornton, and J. E. Rice. ESOP-based Toffoli Gate Cascade Generation. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 206–209, 2007.

[GAJ06] P. Gupta, A. Agrawal, and N. K. Jha. An algorithm for synthesis of reversible logic circuits. *IEEE Trans. on CAD*, 25(11):2317–2329, 2006.

[GK05] R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informaticae*, 66(4):367–395, 2005.

[GK14] S. G. and N. S. Kumar. Design of high speed low power reversible vedic multiplier and reversible divider. *International Journal of Engineering Research and Applications*, 4(9):70–74, 2014.

[GKD06] D. Große, U. Kühne, and R. Drechsler. HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In *ACM Great Lakes Symposium on VLSI*, pages 43–48, 2006.

[GW15] R. D. Gatfane and M. Waje. Design and implementation of low power 32 bit reversible carry skip adder. *International Journal of Science and Research*, 4(7):67–70, 2015.

[GWDD09] D. Große, R. Wille, G. W. Dueck, and R. Drechsler. Exact Multiple-Control Toffoli Network Synthesis With SAT Techniques. *IEEE Trans. on CAD*, 28(5):703–715, 2009.

[Ker04] P. Kerntopf. A New Heuristic Algorithm for Reversible Logic Synthesis. In *Design Automation Conf.*, pages 834–837, 2004.

[Lan61] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. and Develop.*, 5(3):183–191, 1961.

[LP09] M. Lukac and M. Perkowski. Quantum Finite State Machines as Sequential Quantum Circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 92–97, 2009.

[MD04a] D. Maslov and G. W. Dueck. Reversible cascades with minimal garbage. *IEEE Trans. on CAD*, 23(11):1497–1509, 2004.

[MD04b] D. Maslov and G. W. Dueck. Improved Quantum Cost for n-bit Toffoli Gates. *Electronic Letters*, 39(25):1790–1791, 2004.

[MDM05] D. Maslov, G. W. Dueck, and D. M. Miller. Toffoli network synthesis with templates. *IEEE Trans. on CAD*, 24(6):807–817, 2005.

[ME13] P. Moallem and M. Ehsanpour. A novel design of reversible multiplier circuit. *International Journal of Engineering, Transactions C: Aspects*, 26(6):577–586, 2013.

[MMD03] D. M. Miller, D. Maslov, and G. W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Design Automation Conf.*, pages 318–323, 2003.

[MWD10] D. M. Miller, R. Wille, and R. Drechsler. Reducing reversible circuit cost by adding lines. In *Int'l Symp. on Multi-Valued Logic*, pages 217–222, 2010.

[MYMD05] D. Maslov, C. Young, D. M. Miller, and G. W. Dueck. Quantum circuit simplification using templates. In *Design, Automation and Test in Europe*, pages 1208–1213, 2005.

[NC00] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[NHJ+09] N. M. Nayeem, M. A. Hossain, L. Jamal, , and H. Babu. Efficient Design of Shift Registers Using Reversible Logic. In *Int'l Conf. on Signal Processing Systems*, pages 474–478, 2009.

[PF96] P. Patra and D. Fussell. On Efficient Adiabatic Design of MOS Circuits. In *Workshop on Physics and Computation*, pages 260–269, Boston, 1996.

[SFWD12] M. Soeken, S. Frehse, R. Wille, and R. Drechsler. RevKit: An Open Source Toolkit for the Design of Reversible Circuits. In *Reversible Computation 2011*, volume 7165 of *Lecture Notes in Computer Science*, pages 64–76, 2012. RevKit is available at www.revkit.org.

[Sin07] C. Sinz. Visualizing sat instances and runs of the dpll algorithm. *Journal of Automated Reasoning*, 39(2):219–243, 2007.

[SM11] M. Saeedi and I. L. Markov. Synthesis and optimization of reversible circuits - a survey. *ACM Computing Surveys*, 2011.

[SSP13] A. Shafaei, M. Saeedi, and M. Pedram. Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures. In

Proceedings of the 50th Annual Design Automation Conference, DAC '13, pages 41:1–41:6, New York, NY, USA, 2013. ACM.

[ST13] M. Soeken and M. K. Thomsen. White Dots Do Matter: Rewriting Reversible Logic Circuits. In *Int'l Conf. on Reversible Computation*, pages 196–208, 2013.

[SWD10] M. Soeken, R. Wille, and R. Drechsler. Hierarchical Synthesis of Reversible Circuits Using Positive and Negative Davio Decomposition. In *Int'l Design and Test Workshop*, pages 143–148, 2010.

[SWD11] M. Saeedi, R. Wille, and R. Drechsler. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing*, 10(3):355–377, June 2011.

[SWG+09] A. Sülflow, R. Wille, C. Genz, G. Fey, and R. Drechsler. FormED: A formal environment for debugging. In *University Booth at the Design, Automation and Test in Europe*, 2009.

[TG08] M. K. Thomson and R. Glück. Optimized Reversible Binary-coded Decimal Adders. *J. of Systems Architecture*, 54:697–706, 2008.

[Tho12] M. K. Thomsen. A functional language for describing reversible logic. In *Forum on Specification and Design Languages*, pages 135–142, 2012.

[Tof80] T. Toffoli. Reversible Computing. In *ICALP*, pages 632–644, 1980.

[TS05] H. Thapliyal and M. B. Srinivas. A Beginning in the Reversible Logic Synthesis of Sequential Circuits. In *MAPLD Int'l Conf.*, 2005.

[VSB+01] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883, 2001.

[Wal99] T. Walsh. Search in a small world. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[WD09] R. Wille and R. Drechsler. BDD-based Synthesis of Reversible Logic for Large Functions. In *Design Automation Conf.*, pages 270–275, 2009.

[WD10] R. Wille and R. Drechsler. Effect of BDD Optimization on Synthesis of Reversible and Quantum Logic. *Electronic Notes in Theoretical Computer Science*, 253(6):57–70, 2010.

[WDOGO12] R. Wille, R. Drechsler, C. Oswald, and A. Garcia-Ortiz. Automatic Design of Low-Power Encoders Using Reversible Circuit Synthesis. In *Design, Automation and Test in Europe*, pages 1036–1041, 2012.

[WGT+08] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 220–225, 2008. RevLib is available at www.revlib.org.

[WKD11] R. Wille, O. Keszöcze, and R. Drechsler. Determining the Minimal Number of Lines for Large Reversible Circuits. In *Design, Automation and Test in Europe*, pages 1204–1207, 2011.

[WLD14] R. Wille, A. Lye, and R. Drechsler. Optimal swap gate insertion for nearest neighbor quantum circuits. In *ASP Design Automation Conf.*, pages 489–494, 2014.

[WLR11] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM.

[WOD10] R. Wille, S. Offermann, and R. Drechsler. SyReC: A Programming Language for Synthesis of Reversible Circuits. In *Forum on Specification and Design Languages*, pages 184–189, 2010.

[WSD10] R. Wille, M. Soeken, and R. Drechsler. Reducing the Number of Lines in Reversible Circuits. In *Design Automation Conf.*, 2010.

[WSMD14] R. Wille, M. Soeken, D. M. Miller, and R. Drechsler. Trading off circuit lines and gate costs in the synthesis of reversible logic. *INTEGRATION, the VLSI Jour.*, 47(2):284–294, 2014.

[WSOD13] R. Wille, M. Soeken, C. Otterstedt, and R. Drechsler. Improving the mapping of reversible circuits to quantum circuits using multiple target lines. In *18th Asia and South Pacific Design Automation Conference*, pages 145–150, 2013.

[WSPD12] R. Wille, M. Soeken, N. Przigoda, and R. Drechsler. Exact Synthesis of Toffoli Gate Circuits with Negative Control Lines. In *Int'l Symp. on Multi-Valued Logic*, pages 69–74, 2012.

[WSSD12] R. Wille, M. Soeken, E. Schönborn, and R. Drechsler. Circuit Line Minimization in the HDL-Based Synthesis of Reversible Logic. In *IEEE Annual Symposium on VLSI*, pages 213–218, 2012.

[YG07] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Symp. on Partial evaluation and semantics-based program manipulation*, pages 144–153, 2007.

[ZRK07] Z. Zilic, K. Radecka, and A. Kazamiphur. Reversible circuit technology mapping from non-reversible specifications. In *Design, Automation and Test in Europe*, pages 558–563, 2007.