INTEGRATION, the VLSI journal 53 (2016) 39–53

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Invited paper

SyReC: A hardware description language for the specification and synthesis of reversible circuits

Robert Wille [a,b,*], Eleonora Schönborn [c], Mathias Soeken [b,c], Rolf Drechsler [b,c]

[a] Institute for Integrated Circuits, Johannes Kepler University Linz, A-4040 Linz, Austria
[b] Cyber-Physical Systems, DFKI GmbH, Bibliothekstr. 1, 28359 Bremen, Germany
[c] Institute of Computer Science, University of Bremen, Bibliothekstr. 1, 28359 Bremen, Germany

ARTICLE INFO

Article history:
Received 18 December 2014
Received in revised form
5 October 2015
Accepted 10 October 2015
Available online 14 November 2015

Keywords:
Reversible logic
Hardware description languages
Synthesis
Optimization

ABSTRACT

Although researchers and engineers originally focused on a preponderantly *irreversible* computing paradigm, alternative models receive more and more attention. Reversible computation is a promising example which has applications in many emerging technologies such as quantum computation or alternative directions for low-power design. Accordingly, the design of reversible circuits has become an intensely studied research area. In particular, the efficient synthesis of complex reversible circuits poses an important and difficult research question. Most of the solutions proposed thus far are based on pure Boolean function representations such as truth tables or decision diagrams.

In this paper, we provide a comprehensive introduction to and present extensions for the hardware description language SyReC which allows for the specification and automatic synthesis of reversible circuits. Besides a detailed presentation of the language's concepts and operations, we additionally propose algorithms that optimize the resulting circuits with respect to different objectives. A case study on a RISC CPU as well as a thorough experimental evaluation of both, the synthesis approach and its optimizations, show the applicability and demonstrate the advantage of SyReC compared to other solutions based on Boolean function representations.

© 2015 Published by Elsevier B.V.

1. Introduction

Researchers and engineers have focused the investigation of computing machines on a preponderantly *irreversible* computing paradigm. In fact, most of the established computations are not invertible as a standard logical operation such as an AND illustrates. Although it is possible to obtain the inputs of an AND gate if the output is set to 1 (then, both inputs must be set to 1 as well), it is not possible to uniquely determine the input values if the gate outputs 0. While mainly relying on this *conventional* way of computation, also its alternative reversible paradigm receives increasing attention.

Reversible computation is a computing paradigm which only allows bijective operations, i.e. reversible n-input n-output functions in which no two input vectors are mapped to the same output pattern. Hence, all computations can be reverted as the inputs can be obtained from the outputs *and* vice versa.

Although not so well established thus far, reversible computation enables several promising applications and superiors conventional computation paradigms in many domains. Recent accomplishments and experimental validations in these domains made this alternative paradigm also interesting for computer aided design. In particular applications to low-power design and quantum computation accelerated this development.

Low power computation may significantly profit from reversible circuits because, as observed by Landauer [1], power is always dissipated when information is lost during computation. This indeed happens independently from the applied technology. Hence, all computing machines following the conventional paradigm always lose power if irreversible operations (as the above-mentioned AND) are performed. Although the fraction of dissipated power is negligible today, it will become substantial with the expected ongoing miniaturization. Moreover, Gershenfeld has shown that the actual power dissipation corresponds to the amount of energy used to represent the signal [2]. With ongoing miniaturization of circuits, this amount will soon become substantial. Since reversible computations are information loss-less (inputs can always be restored from the outputs and vice versa),

* Corresponding author at: Institute for Integrated Circuits, Johannes Kepler University Linz, A-4040 Linz, Austria. Tel.: +43 732 2468 4739; fax: +43 732 2468 4735.
E-mail addresses: robert.wille@jku.at (R. Wille), eleonora@informatik.uni-bremen.de (E. Schönborn), msoeken@informatik.uni-bremen.de (M. Soeken), drechsle@informatik.uni-bremen.de (R. Drechsler).

http://dx.doi.org/10.1016/j.vlsi.2015.10.001
0167-9260/© 2015 Published by Elsevier B.V.

this power loss can significantly be reduced or even entirely avoided with the alternative paradigm (e.g. [3,4]). Recently, this *has experimentally been verified* in [5].

Quantum computation [6] allows for breaching complexity bounds which are valid for computing devices based on conventional computing by using quantum mechanical phenomena such as superposition and entanglement. Considering that many of the established quantum algorithms include a significant Boolean component (e.g. the oracle transformation in the Deutsch–Jozsa algorithm [7], the database in Grover's search algorithm [8], and the modulo exponentiation in Shor's algorithm [9]), it is crucial to have efficient methods to synthesize quantum gate realizations of Boolean functions. Since any quantum operation inherently is reversible, reversible circuits can be exploited for this purpose.

Moreover, further promising applications have recently been investigated in the design of *low-power encoding and decoding devices* for on-chip interconnects in systems-on-a-chip [10], for *adiabatic circuits* [11], and for circuits employing *energy recovery logic* [12].

Motivated by these promising developments, design and synthesis of reversible circuits have received significant attention in the last decade. This led to several synthesis techniques following complementary schemes and exploiting different kinds of function representations. Examples include approaches

- relying on a permutation-based [13] or cycle-based [14] representation,
- following a transformation-based scheme [15],
- using functions descriptions such as positive-polarity Reed–Muller expansion [16], Reed–Muller spectra [17], or Exclusive Sum of Products [18], and
- exploiting efficient data-structures such as Binary Decision Diagrams [19] or Quantum Multiple-valued Decision Diagrams [20].

However, all these approaches only allow one to automatically synthesize reversible circuits up to a certain size that is bounded by the underlying function representation. In order to provide an efficient design flow for this kind of computation and its applications, synthesis of reversible logic has to reach a level which allows for the description of circuits at higher levels of abstraction. For this purpose, hardware description languages can be exploited. In conventional synthesis, approaches using languages such as VHDL [21], SystemC [22], or SystemVerilog [23] are used to specify and subsequently synthesize circuits. Even in some of the application domains of reversible logic, namely quantum computation, first programming languages have been proposed (see e.g. Quipper [24] or Scaffold [25] as well as related overviews such as e.g. [26]).

In this work, we provide a comprehensive introduction to and extensions for the hardware description language *SyReC*.[1] SyReC allows for the specification and automatic synthesis of complex reversible circuits. For this purpose, established concepts from the previously introduced reversible software language *Janus* [28,29] are adapted. To the best of our knowledge only two other hardware description languages for reversible logic have been proposed after SyReC has been introduced. In [30] a functional reversible language has been proposed that ensures reversibility using a type system based on linear types. As a consequence, only pure reversible functionality can be addressed, i.e. irreversible operations are not supported. A combinator-style functional language has been presented in [31] which offers the design of reversible circuits at the gate level and therefore focuses on lower levels than SyReC.

[1] A preliminary introduction of SyReC is available in [27].

In the remainder of this paper, the concepts of the SyReC language as well as the corresponding synthesis methodology are described in detail. After a brief review on the basics of reversible logic and circuits, Section 3 introduces the general concept as well as the precise syntax and the (informal) semantics of the proposed language. It is shown how reversibility in the description is ensured while at the same time a wide range of (also non-reversible) functionality can be provided. The realization of a reversible control flow is also discussed. All concepts are illustrated by means of examples. Section 4 describes how the resulting programs are realized as a reversible circuit. A hierarchical synthesis approach is presented that automatically transforms the respective statements and operations of the new language into a reversible circuit. The respective realizations of the individual statements and expressions as building blocks are presented and discussed. While this allows for the synthesis of complex reversible circuits, the quality of the results can still be improved. Sections 5 and 6 introduce optimization techniques which allow for a reduction of the number of lines or gate costs depending on the individual design needs. Finally, the applicability of the language as well as its synthesizer is demonstrated in Section 7. Here, a fully functional RISC CPU is designed and realized using SyReC. This confirms that even complex logic can easily be realized with this language. Furthermore, the effects of the different optimizations are evaluated and the synthesizer is compared to another solution which is based on a Boolean function representation.

Overall, a comprehensive overview on this new hardware description language for reversible circuit design is provided. This eventually lifts the design of circuits following the reversible computation paradigm from the Boolean level to a higher abstraction.

2. Preliminaries

To keep the paper self-contained, this section introduces necessary definitions on reversible logic and circuits.

2.1. Reversible functions

A propositional or Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ over the *variables* $X = \{x_1, ..., x_n\}$ is called *reversible* if it is bijective. Clearly, many Boolean functions of practical interest are not reversible (e.g. the conjunct of two propositional variables with a truth table represented by the bit-string 0001). In order to realize such functionality as a reversible circuit, the corresponding functions are *embedded* [32,33]. This is achieved by adding so-called *garbage outputs* which are used to distinguish equal output patterns, thus making the function injective. As a last step in the embedding *constant inputs* are added to equalize the number of input variables and output variables of the function, thus making it bijective.

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, fanout 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* [34] and the *Fredkin gate* [35]. 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

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

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

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

with $x'_s = \overline{c}' x_s \oplus c' x_t$, $x'_t = \overline{c}' x_t \oplus c' x_s$, 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. $f = f_1 \circ f_2 \circ \cdots \circ f_d$.

In addition to the constant inputs and garbage outputs that are added to a function in the process of embedding, for circuits we also consider 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.

Example 1. Fig. 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 order to measure the costs of a circuit, different metrics can be 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 paper, we use the quantum cost metric presented in [36,37] that grows exponentially with respect to the number of control lines of a gate and transistor costs as presented in [4] that grows linearly with respect to the number of control lines.

3. The SyReC language

In the following, 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 section. Afterwards, the syntax and semantics of all SyReC description means are explained in detail.

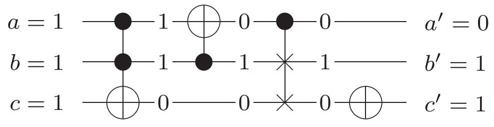


Fig. 1. Reversible circuit.

3.1. General concepts

In order to ensure reversibility in its description, SyReC adapts established concepts from the previously introduced reversible programming language Janus [29] 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.

3.1.1. 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

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

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

3.1.2. 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 ($+, *, /, \%, *>$), bit-wise ($\&, |, \hat{}$), 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. (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. $c\hat{}=(a*b)$ is applied).

3.1.3. 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.

3.1.4. 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.

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⟩ ')')
```

Fig. 2. Syntax of the hardware description language SyReC.

Table 1
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

- 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. 2. In the following, the syntax and the semantics of all description means are explained and illustrated in detail.

3.2. Module and signal declarations

Each SyReC specification (denoted by ⟨*program*⟩ in Line 1 in Fig. 2) 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 (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[2] (denoted by *state*; Line 5). The access modifier affects properties in the synthesized circuits as summarized in Table 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 2. Fig. 3 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

[2] 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 [38,39].

```
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
```

Fig. 3. Exemplary module declarations.

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

Fig. 4. Calling a module identified by adder1.

given by a 4-segment array composed of 16-bit signals (adder3), and an arbitrary module with local and state signals (myCircuit).

3.3. 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 $\langle signal \rangle$ 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).

3.3.1. 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 3. If a SyReC description of an adder is available (as e.g. declared in Fig. 3), it can be added to a design by the call command as shown in Fig. 4.

3.3.2. 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 4. Fig. 5 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.

3.3.3. 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

a

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

Simple loop

b

```
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
```

Loop over bits of a signal

c

```
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
```

Loop with *step* keyword

Fig. 5. Loops in SyReC. (a) Simple loop, (b) loop over bits of a signal, and (c) loop with *step* keyword.

```
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))
```

Fig. 6. Conditional statements in SyReC.

Table 2
Statements in SyReC.

Operation	Semantic
$x \mathrel{\hat{=}} e$	Bit-wise 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. $x{:=}x-e$
$= x$	Bit-wise inversion of x
$++= x$	Increment of x
$--= x$	Decrement of x
$x <=> y$	Swapping value of x with value of y

else-bock modifies an input value of the conditional expression, the *if*- and the *fi*-expression are identical.

Example 5. Fig. 6 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.

3.3.4. Assignment statements

All further statements include the reversible assignment statements (denoted by $\langle assign{-}statement \rangle$), unary statements (denoted by $\langle unary{-}statement \rangle$), and the swap statement (denoted by $\langle swap{-}statement \rangle$) as defined in Lines 12–14. The semantics of these statements is summarized in Table 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.

```
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
```

Fig. 7. Assignment, unary, and swap statements in SyReC.

Example 6. Fig. 7 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.

3.4. Expressions

Expressions as defined in Lines 17–20 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 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 7. Fig. 8 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 $c\hat{=}a{*}b$). It can easily be seen that despite the usage of non-reversible operations in Fig. 8, all statements still can be executed in both directions.

Using the language introduced in this section, 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 Section 7 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.

Table 3
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	Bit-wise XOR of e and f
e & f	Bit-wise AND of e and f
$e \mid f$	Bit-wise OR of e and f
e	Bit-wise inversion of e
e && f	Logical AND of e and f
$e \Vert f$	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
$e \mathrel{!=} f$	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
$e \lessdot N$	Logical left shift of e by N
$e \gtrdot N$	Logical right shift of e by N

```
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)
```

Fig. 8. Application of expressions.

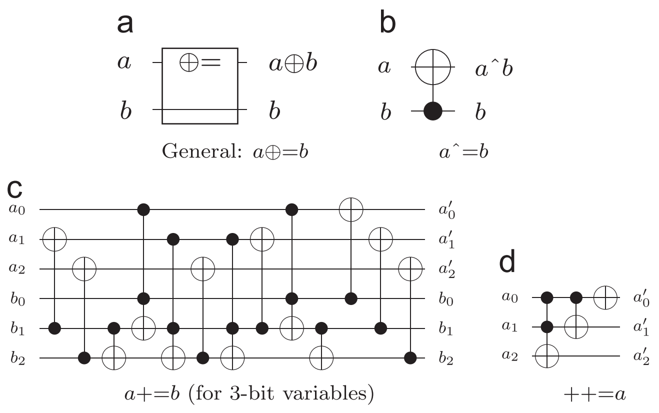


Fig. 9. Synthesis of assignment statements.

4. Synthesis of SyReC specifications

In order to synthesize a given SyReC specification, we developed a hierarchical synthesis method that uses existing realizations, so-called *building blocks*, of the individual statements and expressions and combines them so that the desired circuit results. More precisely, our approach (1) traverses the whole program and (2) adds cascades of reversible gates to the circuit realizing each statement or expression.

Modules are synthesized independently of each other and afterwards cascaded according to the respective *call* and *uncall*-statements. All signals are realized by buses of common 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.

4.1. Synthesis of assignment statements

As introduced in Section 3.3.4, assignment statements in SyReC must be reversible. As a consequence signal values are not overwritten but rather updated with a new value such that the old value can still be recovered by applying the inverted assignment operation. In the following, we use the notation depicted in Fig. 9(a) 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 bit-wise XOR (e.g. $a\hat{=}b$). For 1-bit signals, this operation can be synthesized by a single Toffoli gate as shown in Fig. 9(b). If signals with a bit-width greater than 1 are applied, for each bit a Toffoli gate is applied analogously.

To synthesize the increase operation (e.g. $a+=b$), a modified addition network is added. In the past, several realizations of addition in reversible logic have been investigated. In particular, it is well known that the minimal realization of a one-bit adder consists of four Toffoli gates (e.g. [40]). Thus, cascading the required number of one-bit adders is a possible realization. But since every one-bit adder also requires one constant input, this is a very poor solution with respect to circuit lines. In contrast, heuristic realizations exist that require a fewer number of additional lines (e.g. [41]). Since the increase operation, unlike the addition in general, is reversible, it can even be synthesized without any additional lines. Such a realization is used in our approach. A corresponding cascade for a 3-bit increase is depicted in Fig. 9(c).

The mapping for the decrease operation is left (e.g. $a-=b$). Since the decrease operation is the inverse of the increase operation, the same realization as depicted in Fig. 9(c) is used in reverse.

Finally, the realizations for the unary and swap statements are straight-forward. The bit-wise inversion (e.g. $\sim=a$) is realized by adding a NOT gate to each circuit line representing a bit of *a*. Similarly, a swap (e.g. $a<=>b$) is realized by adding a SWAP gate to the corresponding circuit lines of *a* and *b*. To synthesize an increment (e.g. $++=a$), a cascade as depicted in Fig. 9(d) is applied. A decrement (e.g. $--=a$) is realized by using the same cascade in reverse.

4.2. Synthesis of expressions

Expressions include operations that are not necessarily reversible so that their inputs have to be preserved to allow a (reversible) computation in both directions. To denote such operations, in the following the notation depicted in Fig. 10(a) is used. Again, solid lines represent the signals(s) whose values are preserved (i.e. in this case the input signals).

Synthesis of irreversible functions in reversible logic is not new so that for most of the respective operations reversible circuit realizations already exist. Additional lines with constant inputs are applied to make an irreversible function reversible (e.g. [32,33]). As an example, Fig. 10 shows a reversible gate that realizes an AND operation. As can be seen, this requires one additional circuit line with a constant input 0. Similar mappings exist for all other operations.

Since expressions can be applied together with assignment statements (e.g. $c\hat{}=a\&b$), sometimes a more compact realization is possible. More precisely, additional (constant) circuit lines can be saved (at least for some statements), if the result of an expression is applied to an assignment statement. As an example, Fig. 10(c) shows the realization for $c\hat{}=a\&b$ where no constant input is needed since the circuit line representing *c* is used instead. However, such a simple "combination" is not possible for all statements. As an example, Fig. 10(d) shows a two-bit addition whose result is applied to a bit-wise XOR, i.e. $c\hat{}=a+b$. 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 at the lines representing the sum. Since these values are reused later, performing the XOR operation "in parallel" would destroy the result. Thus, to have a combined realization of a bit-wise XOR and an addition, a precise embedding for this case must be generated. Since determining the respective embeddings and circuits for arbitrary combinations of statements and expressions is a cumbersome task, constant lines are applied to realize the respective functionality. However, later in Section 5 an extended synthesis scheme that removes many of these additional lines is presented.

4.3. Synthesis of the control logic

Finally, the synthesis of control logic is considered. Module calls/uncalls and loops are realized in a straightforward manner. More precisely, loops are realized by simply cascading (i.e. unrolling) the respective statements within a loop block for each iteration. Since the number of iterations must be fixed (cf. Section 3.3.2), this results in a finite number of statements which are subsequently processed. Call and uncall of modules are handled similarly. Here, the respective statements in the modules are cascaded.

To realize conditional statements (i.e. *if*-statements as introduced in Section 3.3.3), two complementary variants are proposed. The first one is depicted in Fig. 11(b). 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

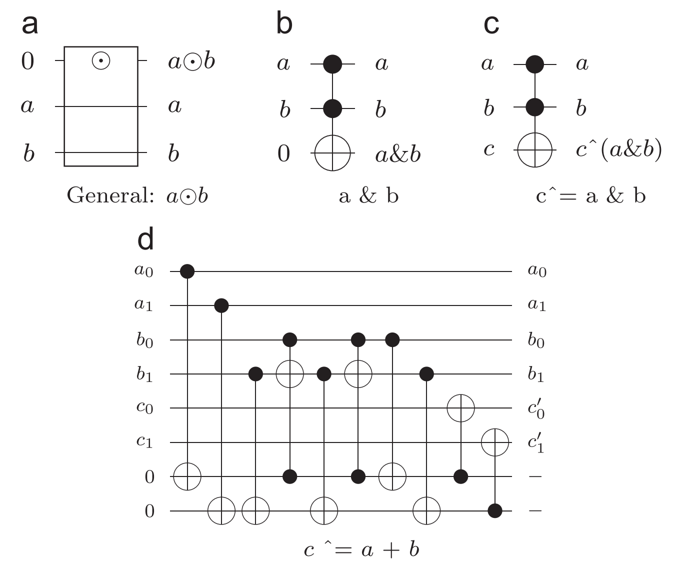


Fig. 10. Synthesis of expressions.

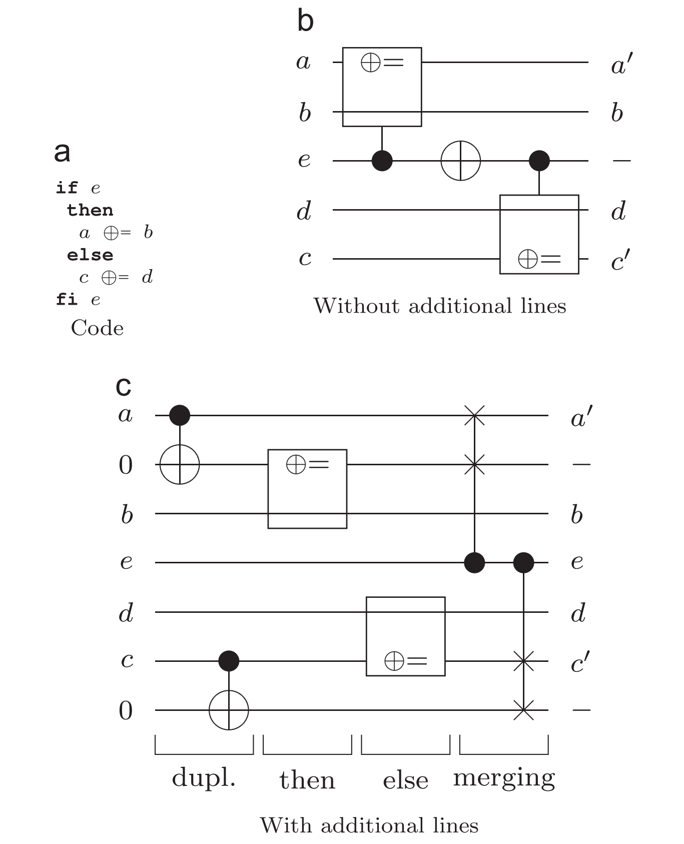

```
if e
 then
   a ⊕= b
 else
   c ⊕= d
fi e
```

Fig. 11. Synthesis of conditional statements. (a) Code. (b) Without additional lines. (c) With additional lines.

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.

Fig. 11(c) 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. 11(a) 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. This, and other aspects, are further evaluated in Section 7.

4.4. Summary

Using the building blocks for statements and expressions as introduced above, it is possible to automatically synthesize reversible circuits specified in SyReC. More precisely, 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. 12, i.e. cascades of building blocks for the respective assignment statements and their expressions result.

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.

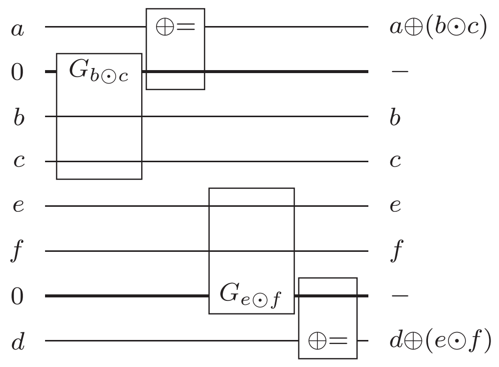


Fig. 12. Resulting circuit structure.

5. Line-aware synthesis of SyReC specifications

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 [42]). The idea is to use the same building blocks as introduced in the previous section, 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 [43]. This enables that circuit lines which have been occupied by expressions before to be reused.

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 number of additional circuit lines is discussed.

5.1. General concept

The extended synthesis approach follows the scheme as introduced in the end of the previous section, 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 8. Consider the two following generic HDL statements:

a $\oplus$ =(b $\odot$ c); d $\oplus$ =(e $\odot$ f);

Fig. 13 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 Section 4 and leads to additional lines with constant inputs (highlighted thick). Afterwards, a further sub-circuit $G_{b\odot c}^{-1}$ is applied. Since $G_{b\odot c}^{-1}$ 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. 13.

5.2. 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.

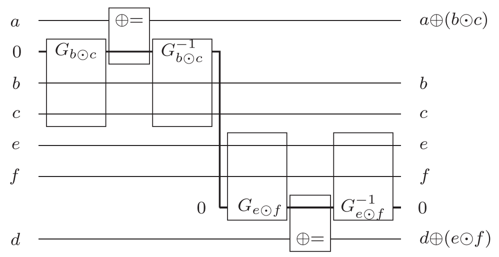


Fig. 13. Line reduction.

5.2.1. *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 arranged as already illustrated in Fig. 13. 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 9. Fig. 14 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.

5.2.2. *Conditional statements*

As described in Section 4.3, there are two proposed realizations for conditional statements (cf. Fig. 11).

Fig. 15(b) 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. 11(b)). 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*-expression 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 4.3. 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 3.3.3, 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. 15(c) 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. 11(c)). The gates highlighted in dark gray (light gray) correspond to the *then*-block (*else*-block). Also, 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. 15(c)). The conditional statement is then realized by applying the procedure described in Section 4.3. 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 on the value of e (e.g. a if $e=1$ and a' if $e=0$) and the realizations of the *then*- and the *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.

5.2.3. *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 the finite number of iterations.

5.3. *Discussion*

Applying the extended synthesis scheme, every statement is synthesized with zero garbage outputs and only additional ancilla

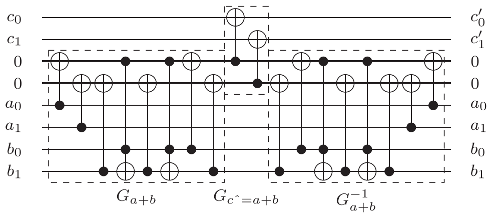


Fig. 14. Synthesizing $c\hat{}=(a+b)$.

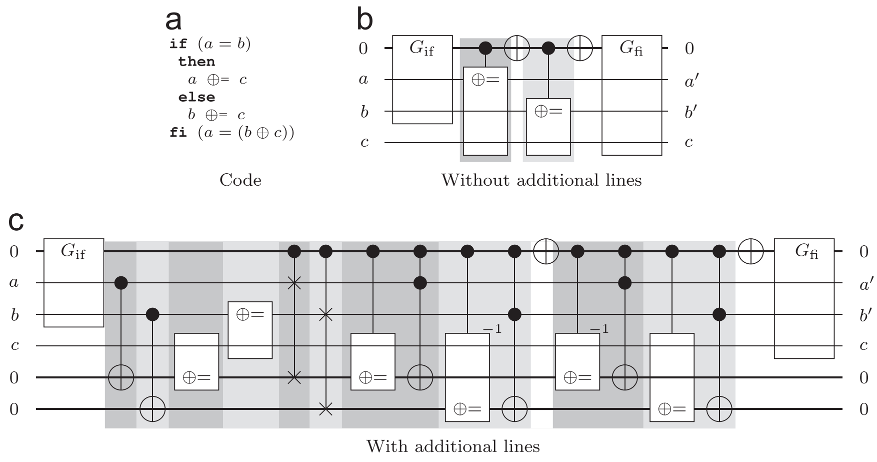


Fig. 15. Synthesizing conditional statements. (a) Code. (b) Without additional lines. (c) With additional lines.

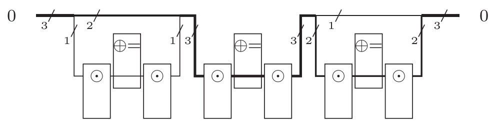


Fig. 16. Effect of expression size.

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 10. 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. 16 illustrates how these circuit lines are applied. For comparison, the synthesis scheme from Section 4 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 11. 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:

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

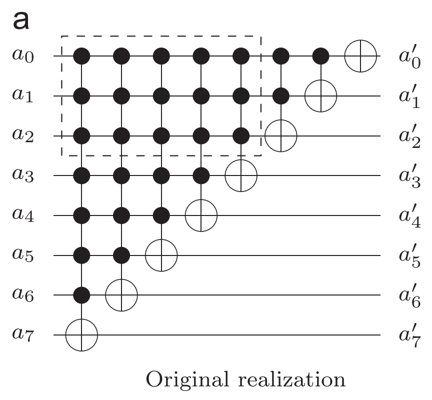
Here, the respective binary operations are applied separately with an assignment operation. Hence, no more than 32 ancilla lines are needed to buffer the intermediate results.

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 Section 4 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, previously, the synthesis of complex functionality as a reversible circuit with the minimal number of lines is a cumbersome task (e.g. [33]), the proposed solution enables to keep this number relatively small.

6. 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 [44] 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 2, the costs of single gates mainly



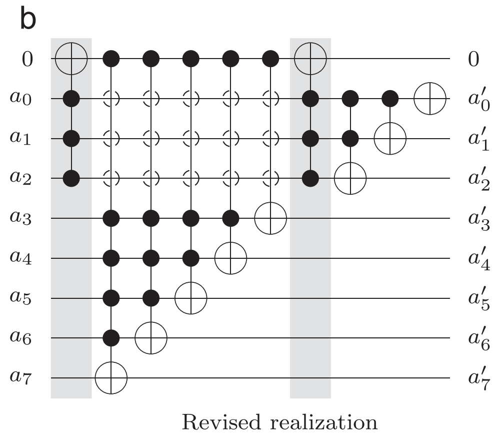


Fig. 17. Cost-aware synthesis. (a) Original realization. (b) Revised realization.

depend on their respective number of control lines. Hence, buffering the results of common control conditions of a cascade of gates allows for reducing 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 12. Consider an 8-bit realization of the increment statement $(++=a)$ as shown in Fig. 17(a). The gates in this cascade have several common control lines, e.g. $C' = \{a_0, a_1, a_2\}$. By adding two Toffoli gates $T(C', \{h\})$, the result of the common control conditions C' can be buffered in an ancilla line h as shown in Fig. 17(b) (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. 17(b), 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.

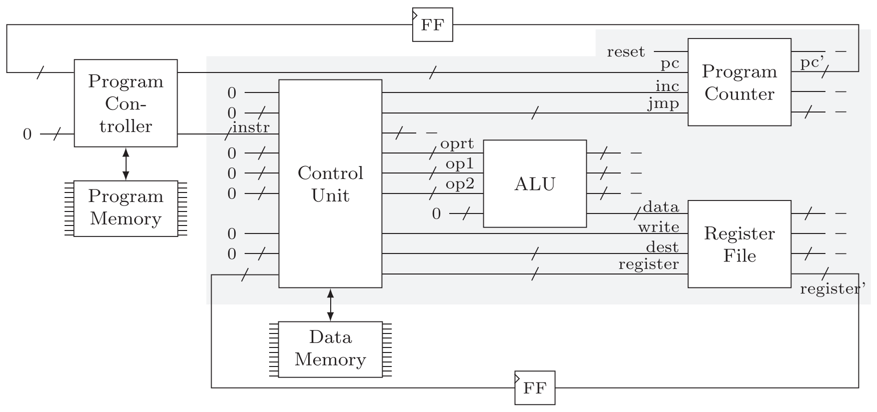


Fig. 18. Schematic diagram of the CPU implementation.

```
module pc(in reset(1), in inc(1), in jmp(11),
          inout pc(11))
  wire zero(11)

  if reset then
    pc <=> zero
  else
    if inc then
      ++= pc
    else
      pc <=> jmp
    fi inc
  fi reset
```

Fig. 19. Implementation of the program counter.

```
0x0   LDL R[7], 4
0x1   LDL R[2], 1
0x2   LDL R[3], 1

      loop:

0x3   ADD R[4], R[3], R[2]
0x4   OR  R[2], R[3], R[0]
0x5   OR  R[3], R[4], R[0]
0x6   SUB R[7], R[7], R[1]
0x7   JNZ loop
```

Fig. 20. Assembler program for Fibonacci number computation.

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 $t_1(C_1, T_1)$ ... $t_k(C_k, T_k)$ 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 $1 \le i \le k$.
   (b) The values of the common control lines are not modified within this cascade, i.e. $C' \cap T_i = \varnothing$ for $1 \le i \le k$.
3. Create a new cascade $\mathrm{T}(C', \{h\})$ $t_1((C_1 \backslash C') \cup \{h\}, T_1)$ ... $t_k((C_k + 1 \backslash C') \cup \{h\}, T_k) \mathrm{T}(C', \{h\})$.
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 Section 4 as well as to the extended scheme proposed in Section 5. 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.

7. Case study and experimental evaluation

SyReC as proposed above enables the design and synthesis of complex logic as a reversible circuit. This has been demonstrated by means of a case study, i.e. the design of a RISC CPU. In this section, the design issues as well as the application of the result are summarized and discussed.

Furthermore, we conducted a thorough study on the effects of the respective optimization approaches presented in Sections 5 and 6. For this purpose, we used the components of the designed CPU as well as further SyReC specifications as benchmarks. The results of this evaluation are also summarized in this section.

7.1. Design of a RISC CPU

To demonstrate the applicability and the benefits of the proposed hardware description language and its synthesizer, the design of a CPU is suitable. A CPU provides a well-known piece of hardware but remains complex enough to bring previously proposed synthesis approaches as discussed in Section 1 to their limits. In this case study, the basic design steps as well as the results are summarized. For a detailed consideration of this experiment, the reader is referred to [45].

The specification of the CPU is inspired by the design of a conventional CPU (cf. [46]). The CPU was created in order to execute software programs provided in terms of an assembler language which includes arithmetic, logic, jump, and load/store instructions. The CPU has been designed as a Harvard architecture with a bit-width of 16 bit for both the program memory and the data memory. The CPU has 8 registers, the size of the program memory is 4 kByte, and the size of the data memory is 128 kByte.

Fig. 18 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. 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

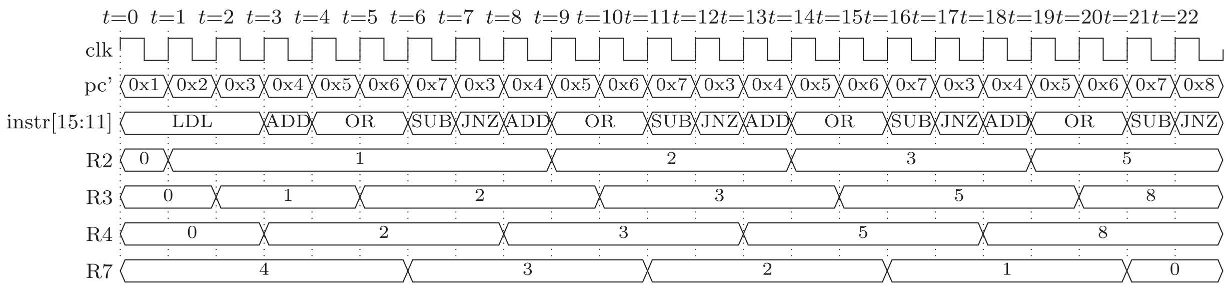


Fig. 21. Waveform illustrating the execution of the program given in Fig. 20.

Table 4
Comparison to previous work.

Benchmark	Bit-width	PI/PO	BDD-based synth. [19]				SyReC synth. if-stm. without add. Lines			SyReC synth. if-stm. with add. Lines		
CPU from Section 7.1			Add. lines	QC	TC	Run-time	Add. lines	QC	TC	Add. lines	QC	TC
cpu_alu	16	55	1852	20,660	77,704	165.99	349	662,531	568,328	2085	31,244	67,896
cpu_alu	32	103	–	–	–	> 500	653	2,235,491	1,917,448	6101	112,396	218,680
cpu_control_unit	16	233	618	7119	27,264	0.12	158	40,433	43,888	413	22,343	31,432
cpu_pc	11	24	39	392	1456	0.00	13	857	912	68	797	1336
cpu_register	16	149	512	7,040	25,600	0.05	18	9833	8472	162	7,560	8472
cpu_register	32	293	1024	14,080	51,200	0.21	34	19,641	16,792	322	15,096	16,920
Benchmarks from RevLib [40]												
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	6275	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	1468	42,281	87,096
callif	16	33	499	7031	26,128	3.80	1	1522	3816	33	641	2664
callif	32	65	–	–	–	> 500	1	3154	7912	65	1313	5480
mult_stmts	16	96	–	–	–	> 500	32	6122	16,960	32	6122	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	6982	11,000	99	1475	5848
nestedif	32	66	–	–	–	> 500	3	14,470	22,776	195	3011	11,992
nestedif2	16	34	257	3348	12,312	1.72	4	8423	8856	100	6034	6824
nestedif2	32	66	–	–	–	> 500	4	31,703	27,736	196	26,674	23,784
varops	16	48	–	–	–	> 500	64	1361	6512	64	1361	6512
varops	32	96	–	–	–	> 500	128	2801	13,424	128	2801	13,424

reset, i.e. the program counter is set to 0. The updated value of the program counter is used in the next cycle.

The SyReC language has been applied to realize all combinational components of this CPU.[3] Being restricted to a reversible language has an impact on the design phase and requires the integration of new design patterns during the implementation. As an example, one new design paradigm becomes already evident in the implementation of the program counter whose SyReC code is given in Fig. 19. 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, this is realized by a new additional signal (denoted by `zero` and set to 0) as well as a swapping operation (Line 6 of Fig. 19). 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). In a similar fashion all remaining components of the CPU are realized. All resulting SyReC codes are available at RevLib [40].

Using the synthesized realizations of these components and plugging them together yields a reversible circuit which is able to process assembler programs as e.g. shown in Fig. 20 (for Fibonacci number computation). Each assembler instruction is translated into a sequence of respective instruction words by applying techniques proposed in [46]. Afterwards, the resulting instruction words are loaded into the program memory, while the data memory is initialized with desired values. Overall, this supports running the translated object code.

Running the program from Fig. 20 on the designed and synthesized CPU yields the waveform given in Fig. 21. 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, respectively. Furthermore, the values of the used registers are provided. 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 Fig. 20. As can be seen, the CPU processes this program as intended. Using SyReC, the design of this complex

[3] Note that the sequential elements, i.e. the memories, are realized by an external controller; here emulated by a Python script.

Table 5
Effect of line- and cost-aware synthesis.

Benchmark	Bit-width	Line-aware synth. (Sect. V)			Cost-aware synth. (Sect. VI) if-stm. without add. Lines			if-stm. with add. Lines			Cost-aware + Line-aware synth.		
		a.l.	QC	TC	a.l.	QC	TC	a.l.	QC	TC	a.l.	QC	TC
CPU from Section 7.1													
cpu_alu	16	87	1,281,717	1,103,200	350	63,025	112,048	2086	30,208	67,144	88	118,751	215,568
cpu_alu	32	151	4,381,653	3,766,496	654	178,783	337,000	6102	107,136	215,112	152	331,151	648,208
cpu_control_unit	16	57	80,142	87,176	159	10,513	23,808	414	7463	21,448	58	20,756	47,304
cpu_pc	11	13	865	944	14	505	672	69	609	1224	14	513	704
cpu_register	16	17	9848	8512	19	2217	3352	163	2600	5144	18	2232	3392
cpu_register	32	33	19,656	16,832	35	3577	5528	323	4760	9496	34	3592	5568
Benchmarks from RevLib [40]													
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	8975	21,088	115	6275	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	1469	24,569	70,968	298	25,950	74,296
callif	16	1	1524	3824	1	1522	3816	33	641	2664	1	1524	3824
callif	32	1	3156	7920	1	3154	7912	65	1313	5480	1	3156	7920
mult_stmts	16	16	11,572	30,704	32	6122	16,960	32	6122	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	6996	11,056	4	3094	7800	99	1475	5848	3	3108	7856
nestedif	32	2	14,484	22,832	4	6358	15,992	195	3011	11,992	3	6372	16,048
nestedif2	16	3	8504	9072	5	5243	6568	101	3809	5224	4	5324	6784
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	2032	9680	64	1361	6512	64	1361	6512	48	2032	9680
varops	32	96	4176	19,920	128	2801	13,424	128	2801	13,424	96	4176	19,920

Table 6
Average values of the respective metrics for all schemes.

	add.lines	QC	TC
Initial approach (Section 4)			
if-stm. w/o additional lines	120.0	286,037.4	252,612.7
if-stm. w/ additional lines	559.1	129,734.5	131,327.3
Line-aware scheme (Section 5)			
	48.8	558,967.7	490,699.7
Cost-aware scheme (Section 6)			
if-stm. w/o additional lines	120.0	34,178.8	67,533.0
if-stm. w/ additional lines	559.7	27,271.7	58,410.7
Cost- & Line-aware scheme			
	49.5	63,610.2	126,376.0

circuitry was significantly easier than with pure Boolean function representations.

7.2. *Evaluation of the resulting circuits*

Besides the case study on the applicability of the hardware description language, 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* [47]. 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* [40]. 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.

7.2.1. *Comparison to previous work*

In a first evaluation, we compared the quality of the circuits obtained using the initial synthesis scheme (as introduced in Section 4) to previously proposed solutions. As discussed in Section 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 [19] 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 [19].

The results are summarized in Table 4. 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*, *Bit-width*, and *PI/PO*, respectively). The following columns give the number of additional circuit lines (*add. lines*), 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. 11(b) (denoted by *if-stm. without add. lines*) and according to Fig. 11(c) (denoted by *if-stm. 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 a BDD exists. Thus, for these components the BDD-based approach suffers from memory explosion.

Besides that, these results also confirm the discussion from Section 4 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.

7.2.2. *Effect of line- and cost-aware synthesis*

In a second evaluation, the effect of the optimized synthesis schemes presented in Section 5 (for line-aware synthesis) and Section 6 (for cost-aware synthesis) has been evaluated. Here, Table 5 presents the results generated with the following schemes:

- The synthesis scheme as described in Section 5 using the realization of *if*-statements according to Fig. 11(b) (denoted by *Line-aware synth*).[4]
- The synthesis scheme as described in Section 6 using the realization of *if*-statements according to Fig. 11(b) (denoted by *Cost-aware synth; if-stm. without add. lines*).
- The synthesis scheme as described in Section 6 using the realization of *if*-statements according to Fig. 11(c) (denoted by *Cost-aware synth; if-stm. with add. lines*).
- The synthesis scheme as described in Sections 5 and 6 combined together with the realization of if-statements according to Fig. 11(b) (denoted by *Cost-aware* + *Line-aware synth.*).

Beyond that, Table 5 uses the same denotation as Table 4. 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 6.

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. 11(c)). 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 is applied. However, the two metrics are complementary to each other. 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 [48].

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) (twice than the best result) are achieved on average.

[4] Note that a realization of if-statements according to Fig. 11(c) has not been considered for this scheme since, as discussed in Section 5.2.2, line-aware synthesis would always lead to an increase in both, additional lines and costs, in this case.

8. Conclusions

In this paper, we investigated, extended, and evaluated the reversible hardware description language SyReC. Besides new syntactical features, two optimization approaches have been proposed that can be applied to reduce synthesis results based on the designer's individual needs. An adjusted synthesis scheme “uncomputes” intermediate results and therefore allows one to keep the number of additional lines small. A second optimization scheme adds a new line which is then used in order to buffer intermediate values. This allows for a reduction of the size of individual gates and, by this, improves the costs of the circuit.

In an experimental evaluation we first demonstrated the applicability by means of a case study in which a RISC CPU has been designed using the hardware description language. Furthermore, SyReC's advantage compared to previously proposed synthesis approaches based on Boolean function representations has been shown. Finally, a closer examination has been made for the two optimization schemes. The results showed that a combination between the line- and costs-aware synthesis scheme provides a good trade-off and, hence, leads to a significantly more efficient compromise.

Future work will focus on the development of strategies which further reduce the resulting costs as well as the number of lines in the resulting circuits. This includes the consideration of more efficient building blocks as well as schemes which reduce the number of required building blocks (in particular the ones for “uncomputing”). In parallel, solutions for code optimization, e.g. term rewriting techniques that best exploit the potential of the proposed synthesis method, shall be investigated. Further ideas addressing these objectives have recently been proposed e.g. in [49].

Acknowledgments

This work has partially been supported by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy, and by the European Union through the COST Action IC1405.

References

[1] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Dev 5 (1961) 183.
[2] N. Gershenfeld, Signal entropy and the thermodynamics of computation, IBM Syst. J. 35 (3–4) (1996) 577–586.
[3] C.H. Bennett, Logical reversibility of computation, IBM J. Res. Dev. 17 (6) (1973) 525–532.
[4] B. Desoete, A.D. Vos, A reversible carry-look-ahead adder using control gates, INTEGRATION, VLSI J. 33 (1-2) (2002) 89–104.
[5] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, E. Lutz, Experimental verification of Landauer's principle linking information and thermodynamics, Nature 483 (2012) 187–189.
[6] M. Nielsen, I. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2000.
[7] D. Deutsch, R. Jozsa, Rapid solution of problems by quantum computation, Proc. R. Soc. Lond. A 439 (1992) 553–558.
[8] L.K. Grover, A fast quantum mechanical algorithm for database search, Theory Comput. (1996) 212–219.
[9] P.W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, Found. Comput. Sci. (1994) 124–134.
[10] R. Wille, R. Drechsler, C. Oswald, A. Garcia-Ortiz, Automatic design of low-power encoders using reversible circuit synthesis, In: Design, Automation and Test in Europe, 2012, p. 1036–1041.
[11] P. Patra, D. Fussell, On efficient adiabatic design of MOS circuits, In: Workshop on Physics and Computation, Boston, 1996, pp. 260–269.
[12] J. Lim, D.-G. Kim, S.-I. Chae, nMOS reversible energy recovery logic for ultra-low-energy applications, J. Solid-State Circuits 35 (6) (2000) 865–875.
[13] V.V. Shende, A.K. Prasad, I.L. Markov, J.P. Hayes, Synthesis of reversible logic circuits, IEEE Trans. CAD 22 (6) (2003) 710–722.

[14] M. Saeedi, M.S. Zamani, M. Sedighi, Z. Sasanian, Reversible circuit synthesis using a cycle-based approach, J. Emerg. Technol. Comput. Syst. 6 (4), 2010.
[15] D. Maslov, G.W. Dueck, D.M. Miller, Toffoli network synthesis with templates, IEEE Trans. CAD 24 (6) (2005) 807–817.
[16] P. Gupta, A. Agrawal, N.K. Jha, An algorithm for synthesis of reversible logic circuits, IEEE Trans. CAD 25 (11) (2006) 2317–2330.
[17] D. Maslov, G.W. Dueck, D.M. Miller, Techniques for the Synthesis of Reversible Toffoli Networks, ACM Trans. Des. Autom. Electron. Syst. 12 (4), 2007.
[18] K. Fazel, M. Thornton, J. Rice, ESOP-based Toffoli gate cascade generation, in: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2007, pp. 206–209.
[19] R. Wille, R. Drechsler, BDD-based Synthesis of reversible logic for large functions, In: Design Automation Conference, 2009, pp. 270–275.
[20] M. Soeken, R. Wille, C. Hilken, N. Przigoda, R. Drechsler, Synthesis of reversible circuits with minimal lines for large functions, In: ASP Design Automation Conference, 2012, pp. 85–92.
[21] R. Lipsett, C. Schaefer, C. Ussery, VHDL: Hardware Description and Design, Kluwer Academic Publishers, Intermetrics, Inc., 1989.
[22] T. Grötker, S. Liao, G. Martin, S. Swan, System Design with SystemC, Kluwer Academic Publishers, 2002.
[23] S. Sutherland, S. Davidmann, P. Flake, System Verilog for Design and Modeling, Kluwer Academic Publishers, 2004.
[24] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, B. Valiron, Quipper: a scalable quantum programming language, In: Conference on Programming Language Design and Implementation, 2013, pp. 333–342.
[25] A.J. Abhari, A. Faruque, M.J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, F. Chong, M. Martonosi, M. Suchara, K. Brown, M. Pedram, T. Brun, Scaffold: Quantum Programming Language, ⟨http://ftp://ftp.cs.princeton.edu/techreports/2012/934.pdf⟩, 2012.
[26] S.J. Gay, Quantum programming languages: survey and bibliography, Math. Struct. Comput. Sci. 16 (4) (2006) 581–600.
[27] R. Wille, S. Offermann, R. Drechsler, SyReC: a programming language for synthesis of reversible circuits, Forum Specif. Des. Lang. (2010) 184–189.
[28] C. Lutz, Janus: a time-reversible language, Letter to R. Landauer. ⟨http://www.tetsuo.jp/ref/janus.html⟩, 1986.
[29] T. Yokoyama, R. Glück, A reversible programming language and its invertible self-interpreter, In: Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 2007, pp. 144–153.
[30] M.K. Thomsen, A functional language for describing reversible logic, Forum Specif. Des. Lang. (2012) 135–142.
[31] M.K. Thomsen, Describing and optimising reversible logic using a functional language, Implem. Appl. Funct. Lang. (2011) 148–163.
[32] D. Maslov, G.W. Dueck, Reversible cascades with minimal garbage, IEEE Trans. CAD 23 (11) (2004) 1497–1509.
[33] R. Wille, O. Keszöcze, R. Drechsler, Determining the minimal number of lines for large reversible circuits, In: Design, Automation and Test in Europe, 2011, pp. 1204–1207.
[34] T. Toffoli, Reversible Computing, In: W. de Bakker, J. van Leeuwen (Eds.), Automata, Languages and Programming, Springer, vol. 632, Technical Memo MIT/LCS/TM-151, MIT Lab for Computer Science, 1980.
[35] E.F. Fredkin, T. Toffoli, Conservative logic, Int. J. Theoret. Phys. 21 (3/4) (1982) 219–253.
[36] A. Barenco, C.H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, H. Weinfurter, Elementary gates for quantum computation, Phys. Rev. A 52 (1995) 3457–3467.
[37] D.M. Miller, R. Wille, Z. Sasanian, Elementary quantum gate realizations for multiple-control toffolli gates, In: International Symposium on Multi-Valued Logic, 2011, pp. 288–293.
[38] M.-L. Chuang, C.-Y. Wang, Synthesis of reversible sequential elements, In: ASP Design Automation Conference, 2007, pp. 420–425.
[39] M. Lukac, M. Perkowski, Quantum finite state machines as sequential quantum circuits, In: International Symposium on Multi-Valued Logic 2009, pp. 92–97,.
[40] R. Wille, D. Große, L. Teuber, G. W. Dueck, R. Drechsler, RevLib: an online resource for reversible functions and reversible circuits, In: International Symposium on Multi-Valued Logic, pp. 220–225, RevLib is Available at ⟨http://www.revlib.org⟩, 2008.
[41] Y. Takahashi, N. Kunihiro, A linear-size quantum circuit for addition with no ancillary qubits, Quant. Inf. Comput. 5 (2005) 440–448.
[42] R. Wille, M. Soeken, E. Schönborn, R. Drechsler, Circuit line minimization in the HDL-based synthesis of reversible logic, In: IEEE Annual Symposium on VLSI, 2012, pp. 213–218.
[43] H.B. Axelsen, Clean translation of an imperative reversible programming language, In: International Conference on Compiler Construction, 2011, pp. 144–163.
[44] D.M. Miller, R. Wille, R. Drechsler, Reducing reversible circuit cost by adding lines, In: International Symposium on Multi-Valued Logic, 2010, pp. 217–222.
[45] R. Wille, M. Soeken, D. Große, E. Schönborn, R. Drechsler, Designing a RISC CPU in reversible logic, In: International Symposium on Multi-Valued Logic, 2011, pp. 170–175.
[46] D. Große, U. Kühne, R. Drechsler, HW/SW co-verification of embedded systems using bounded model checking, In: ACM Great Lakes Symposium on VLSI, 2006, pp. 43–48.
[47] M. Soeken, S. Frehse, R. Wille, R. Drechsler, RevKit: an open source toolkit for the design of reversible circuits, In: Reversible Computation 2011, Lecture Notes in Computer Science, vol. 7165, pp. 64–76, RevKit is Available at ⟨http://www.revkit.org⟩, 2012.
[48] R. Wille, M. Soeken, D.M. Miller, R. Drechsler, Trading off circuit lines and gate costs in the synthesis of reversible logic, INTEGRATION, VLSI J. 47 (2) (2014) 284–294.
[49] Z. Al-Wardi, R. Wille, R. Drechsler, Towards line-aware realizations of expressions for HDL-based synthesis of reversible circuits, In: Reversible Computation, 2015.

Robert Wille received the Diploma and Dr.-Ing. degrees in computer science from the University of Bremen, Bremen, Germany, in 2006 and 2009, respectively. From 2006 to 2015, he has been with the Group of Computer Architecture, University of Bremen, and, since 2013, the German Research Center for Artificial Intelligence, Bremen. He was a Lecturer with the University of Applied Science, Bremen, and a Visiting Professor with the University of Potsdam, Potsdam, Germany, and Technical University Dresden, Dresden, Germany. Since 2015, he is full professor at the Johannes Kepler University Linz, Austria. In the nine years of his research activity, he has published over 100 papers in journals and conferences and served in program committees of numerous conferences such as ASPDAC, DAC, and ICCAD. His current research interests include design of circuits and systems for both conventional and emerging technologies with a focus in the domain of verification and proof engines.

Eleonora Schönborn received the Diploma degree in computer science from the University of Bremen, Bremen, Germany, in 2012. From 2012 to 2015 she was with the Group of Computer Architecture, University of Bremen, and the Graduate School System Design, a cooperation of the University of Bremen with the German Research Center for Artificial Intelligence (DFKI) and the German Aerospace Center (DLR). Her research interests include the design and synthesis of reversible circuits and systems.

Mathias Soeken received the Dr.-Ing. degree in computer science from the University of Bremen in 2013. Since 2009, he is with the Group of Computer Architecture at the University of Bremen and, since 2012, with the German Research Center for Artificial Intelligence (DFKI). His research interests are in electronic design automation, formal verification, natural language processing, and circuit complexity. Since 2012, Mathias Soeken teaches graduate courses at the University of Bremen.

Rolf Drechsler received the Diploma and Dr. phil. nat. degrees in computer science from the J. W. Goethe University Frankfurt am Main, Frankfurt am Main, Germany, in 1992 and 1995, respectively. He was with the Institute of Computer Science, Albert-Ludwigs University, Freiburg im Breisgau, Germany and with the Corporate Technology Department, Siemens AG, Munich, Germany. Since October 2001, he has been with the University of Bremen, Bremen, Germany, where he is currently a Full Professor and the Head of the Group for Computer Architecture, Institute of Computer Science. Since 2011, he is also the Director of the Cyber-Physical Systems group at the German Research Center for Artificial Intelligence (DFKI) in Bremen. His research interests include the development and design of data structures and algorithms with a focus on circuit and system design. In these areas, he published more than 250 papers and served in program committees of international conferences such as DAC, DATE, and ICCAD.