Circuit Line Minimization in the HDL-based Synthesis of Reversible Logic

Robert Wille* Mathias Soeken*† Eleonora Schönborn* Rolf Drechsler*†

*Institute of Computer Science
University of Bremen, 28359 Bremen, Germany

†Cyber-Physical Systems, DFKI GmbH
28359 Bremen, Germany

{rwille,msoeken,eleonora,drechsle}@informatik.uni-bremen.de

Abstract—**In the last decade, reversible circuits have been extensively investigated due to their application in emerging areas such as quantum computation or low-power design. In the past, synthesis of reversible circuits was lifted from the Boolean level to approaches exploiting hardware description languages. However, existing HDL synthesizers lead to circuits with a significant number of additional lines.**

In this work, we focus on the reduction of additional circuit lines which are caused by buffering intermediate results. We propose an approach that reuses these lines as soon as the intermediate results are not required anymore. Experiments confirm that this approach decreases the number of circuit lines by up to two orders of magnitude and 60% on average.

I. Introduction

Reversible circuits are computing devices realizing bijections, i.e. one-to-one mappings of the respective input and output values. Reversibility builds the basis for many emerging technologies enhancing or even replacing conventional computing devices in the future.

Quantum circuits [1] are a prominent example. They enable to solve important problems such as factorization or database search significantly faster than their conventional counterparts (see e.g. [2]). In addition, potential benefits of reversible circuits for CMOS technologies are currently being considered. While the ongoing miniaturization and power reduction of non-reversible computing devices eventually will approach fundamental limits (extrapolated from the observations by Gordon Moore and Rolf Landauer; see e.g. [3]), reversible circuits theoretically allow to breach some of these limits. Prototypical realizations of both quantum circuits and reversible CMOS-based circuits (e.g. [4], [5]) have already shown promising results encouraging further research in this area. Besides that, reversible circuits show promising application in the domain of low-power interconnect decoders [6].

Motivated by these achievements, synthesis of reversible circuits is subject of current research. In the past, mainly synthesis approaches based on function representations such as permutations (e.g. [7]), truth tables (e.g. [8]), or positive-polarity Reed-Muller expansion (e.g. [9]) have been proposed. One major objective of these algorithms is to keep the number of circuit lines (i.e. circuit signals) as small as possible. This originates from the fact that in applications such as quantum computation, the circuit lines (i.e. qubits) are considered to be a very restricted resource [1].

However, since the above mentioned approaches are applicable to very small functions only, researchers strived for more scalable solutions. This led to approaches exploiting more compact function representations such as decision diagrams (e.g. [10], [11]) and even a first *Hardware Description Language* (HDL, see [12]). The latter is considered in detail in this work.

In fact, ensuring reversibility in description languages is a crucial task which has been studied for software and hardware languages (see e.g. [13] and [12]). For both, an established paradigm is to distinguish between (1) *reversible assignment operations* which can reversibly modify values and (2) *binary operations* which are not necessarily reversible, but ensure a wide range of language means.

While reversible assignment operations can easily be realized as reversible circuits, the realization of binary operations is cumbersome. In order to maintain the reversibility, circuits realizing binary operations make use of additional lines that buffer intermediate results. Such an effect is even more emphasized in nested expressions, since existing HDL synthesizers realize them separately, which often leads to a significant amount of additional circuit lines (this is covered in more detail in Section II-C).

In this work, we investigate this characteristic of existing HDL synthesizers. Inspired by [14], we propose an approach that reuses additional lines storing intermediate results as soon as they are not required anymore. Additional circuit gates are spent for this purpose. Experiments demonstrate that the proposed approach does not only outperform existing HDL-based synthesis with respect to the number of lines but also post-synthesis optimization methods recently introduced in [15]. In total, the number of lines decreases by up to two orders of magnitude and 60% on average, while the costs due to the additional gates remain acceptable.

The remainder of this paper is organized as follows. The following section reviews the background of this work and motivates the considered research question. Section III provides the general idea of the proposed solution which afterwards is described in detail in Section IV. Then, certain characteristics of the new HDL synthesizer are discussed in Section V. Finally, Section VI provides experimental results while Section VII concludes this paper.

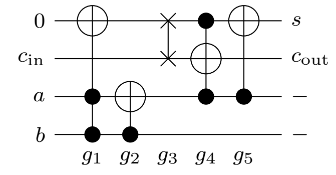


Fig. 1. Reversible circuit realizing a full adder

II. Background and Motivation

This section briefly reviews the basics on reversible circuits, a reversible HDL, as well as the corresponding HDL-based synthesis. It provides the necessary background to keep the paper self-contained and additionally illustrates the problem addressed in this work.

A. Reversible Circuits

Reversible circuits realize functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ with a unique input/output mapping, i.e. bijections. A reversible circuit $G = g_1 \dots g_d$ is composed as a cascade of reversible gates g_i [1]. The inverse of G (representing the function f^{-1} and denoted by G^{-1}) can be obtained by cascading $g_d^{-1} g_{d-1}^{-1} \cdots g_1^{-1}$, where g_i^{-1} is the inverse gate of g_i. Since the self-inverse Toffoli and Fredkin gates are considered in this paper (see below), $g_i = g_i^{-1}$ holds and, thus, G^{-1} can simply be obtained by reversing the order of the gates of G.

For a set of signals $X = \{x_1, \dots, x_n\}$, a *reversible gate* has the form $g(C, T)$, where $C = \{x_{i_1}, \dots, x_{i_k}\} \subset X$ is the set of *control lines* and $T = \{x_{j_1}, \dots, x_{j_l}\} \subseteq X$ with $C \cap T = \emptyset$ is the non-empty set of *target lines*. The gate operation is applied to the target lines if, and only if, all control lines meet the required control conditions. Control lines and unconnected lines always pass through the gate unaltered.

In the literature, several types of reversible gates have been introduced. Usually, circuits realized by *Toffoli gates* and *Fredkin gates* are considered. A Toffoli gate has a single target line x_j and uniquely maps the input $(x_1, x_2, \dots, x_j, \dots, x_n)$ to the output $(x_1, x_2, \dots, x_{i_1} x_{i_2} \cdots x_{i_k} \oplus x_j, \dots, x_n)$. That is, a Toffoli gate inverts the target line if, and only if, all control lines are assigned the logic value 1. A Fredkin gate has two target lines x_{j_1} and x_{j_2} and interchanges their values if, and only if, the conjunction of all control lines evaluates to 1.

By definition, reversible circuits can only realize reversible functions. In order to realize non-reversible functions, *additional circuit lines* with constant inputs and garbage outputs (i.e. don't care outputs) are applied (see e.g. [16], [17]). Furthermore, additional circuit lines are also used frequently in hierarchical synthesis approaches (e.g. [10], [12]).

Example 1: Fig. 1 shows a reversible circuit realization of a 1-bit adder. Black circles represent control lines while $\oplus$ and $\times$ represent the target lines of a Toffoli and Fredkin gate, respectively. Since the adder is a non-reversible function, one additional circuit line is used to realize this function as a reversible circuit. The gates g_1, g_2, g_4, and g_5 are Toffoli gates, while the gate g_3 is a Fredkin gate.

```
1  module simple-alu(in op(2), in a, in b, out c)
2  if (op = 0) then
3    c ^= (a + b)
4  else
5    if (op = 1) then
6      c ^= (a - b)
7    else
8      if (op = 2) then
9        c ^= (a * b)
10     else
11       c ^= (a / b)
12     fi (op = 2)
13   fi (op = 1)
14 fi (op = 0)
```

Fig. 2. SyReC specification of a simple ALU

B. Reversible HDL

A major motivation of research in the domain of reversible circuit synthesis is the striving for better scalability in order to enable the efficient design of complex functionality. Consequently, HDLs became a focus of ongoing research. A first version of an HDL for reversible circuits named *SyReC* has been introduced in [12]. SyReC is based on the reversible software language *Janus* [13], which has been enriched by further concepts (e.g. declaring circuit signals of different bit-widths), new operations (e.g. bit-access and shifts), and some restrictions (e.g. the prohibition of dynamic loops). In the following, we briefly review the main concepts of this HDL by means of Fig. 2 which depicts a SyReC specification of a simple arithmetic logic unit[1].

As can be seen, a SyReC description includes the declaration of modules and signals of the circuit to be specified (Line 1). Signals represent non-negative integers as their sole data type. Furthermore, a variety of statements and expressions are available to specify the functionality of the circuit and in order to ensure reversibility, these statements must satisfy certain criteria. For example, in each conditional statement the *if-expression* has to be terminated by a corresponding *fi-expression* (see e.g. Line 12). Furthermore, statements and expressions are distinguished between *reversible assignment operations* (denoted by $\oplus$=) and not necessarily reversible *binary operations* (denoted by $\odot$).

Reversible assignment operations assign values to a signal on the left-hand side. Therefore, the respective 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 bit-wise 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, bit-wise, logical, or relational operations, may not be reversible. Thus, they can only be used in right-hand expressions which preserve the values of the respective inputs. In doing so, all computations remain reversible since the input values can be applied to reverse any operation. For example, to specify the multiplication in Line 9, a new free signal c in combination with a reversible assignment operation is applied.

[1]For a more detailed treatment, we refer to [12] as well as to the detailed documentation provided at the RevLib benchmark webpage [18].

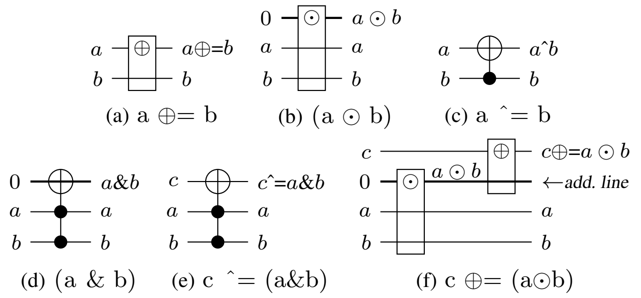


Fig. 3. Circuits obtained by HDL-based synthesis

C. HDL-based Synthesis

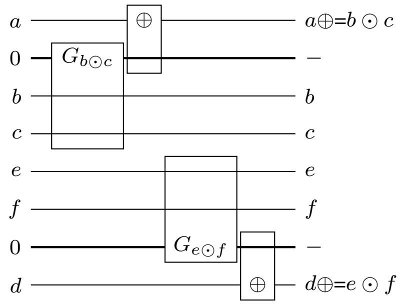
The application of HDLs enables the design of reversible circuits on a higher level. For example, the arithmetic logic unit can be specified much easier using the code from Fig. 2 in comparison to methods based on truth tables or decision diagrams. However, the specified circuits still need to be synthesized. For this purpose, a hierarchical synthesis method is applied [12]. Here, existing realizations of the individual operations (i.e. building blocks) are combined so that the desired circuit is built. For the realization of these building blocks, we refer to previous work (e.g. [19], [20]).

In the following, we use the notation depicted in Fig. 3(a) and Fig. 3(b) to denote a building block for a reversible assignment operation and for a binary operation, respectively. Circuit lines drawn through the blocks represent the signals which values are preserved. Since binary operations represent non-reversible operations, a sole circuit realization usually requires additional circuit lines with constant inputs 0.

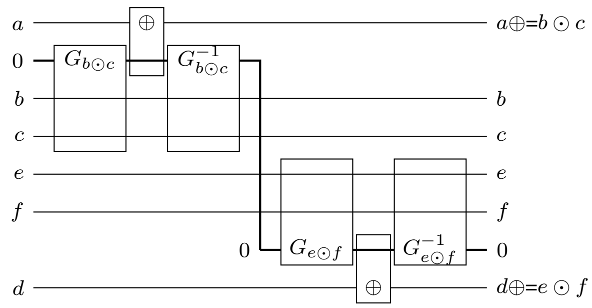
Example 2: Fig. 3(c) shows the realization of a bit-wise XOR (i.e. a ^= b) – the simplest reversible assignment operation. In this case, a bit-width of 1 is assumed for the signals a and b. If signals with a larger bit-width are considered, a Toffoli gate is applied analogously for each bit. Fig. 3(d) shows a realization of an AND operation – a typical binary operation. As the AND operation is non-reversible, an additional circuit line with a constant input is required.

Since binary operations can only be applied in combination with a reversible assignment operation, additional circuit lines are not necessary in general. As an example, Fig. 3(e) shows the realization for c ^= (a & b) where no additional line is applied, but the signal representing c is used instead. However, determining the respective circuits for arbitrary combinations of reversible assignment operations and binary operations is a cumbersome task. Thus, existing HDL synthesizers make use of additional circuit lines where intermediate results of binary operations are buffered before the building block of the corresponding reversible assignment operation is applied. This is illustrated in Fig. 3(f).

This procedure leads to a significant number of additional circuit lines which is commonly seen as disadvantageous. This originates from the fact that in many application areas of reversible circuits (in particular in the domain of quantum computation [1]), the number of circuit lines is a highly limited resource. As a result, keeping the number of circuit lines as small as possible is very important and has hardly been addressed in HDL-based synthesis so far.



(a) Original synthesis



(b) Proposed idea

Fig. 4. General Idea

Motivated by this, in this paper we present a synthesis approach for HDL descriptions of reversible circuits which aims for the determination of efficient realizations with respect to the number of lines.

III. General Idea

In this section, we present the idea behind our approach. The goal is to synthesize HDL descriptions of reversible circuits keeping the number of circuit lines as small as possible. For this purpose, an approach is proposed which realizes each statement in the HDL code in three steps.

1) Compose a sub-circuit $G_\odot$ realizing the right-hand side expression of the statement using the existing building blocks of the binary operations. The result of the expression is buffered by means of additional circuit lines (see Fig. 3(b)).
2) Compose a sub-circuit $G_\oplus$ realizing the overall statement using the existing building blocks of the reversible assignment operation together with the buffered result of the right-hand side expression (see Fig. 3(f)).
3) Add the inverse circuit from Step 1, i.e. $G_\odot^{-1}$, to the circuit in order to set the circuit lines buffering the result of the right-hand side expression back to the constant 0.

Example 3: The general idea is illustrated by means of the following two generic HDL statements:

a ⊕= (b ⊙ c);
d ⊕= (e ⊙ f);

Fig. 4 sketches the resulting circuit after applying the steps outlined above. 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 established procedure from Fig. 3(f) and leads to additional lines with constant inputs (highlighted thick). But in contrast to existing HDL synthesizers, a further

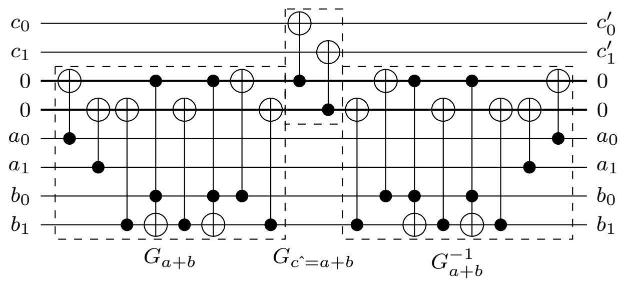


Fig. 5. Synthesizing c ˆ= (a+b)

sub-circuit $G^{-1}_{b\odot c}$ is applied afterwards. 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⊕=$e \odot f$ in Fig. 4(b).

Following this procedure, each statement can be realized with zero garbage outputs. While this sketches the general idea, the implementation for the respective statements is described in the following.

IV. Implementation

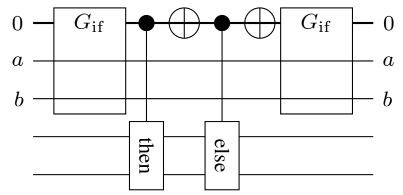
In this section, the proposed approach is described. The general structure of our HDL synthesizer follows the basic concepts of [12]. That is, a hierarchical synthesis method is implemented that traverses each statement of the HDL description and applies existing building blocks of the individual operations in order to combine them so that the desired circuit is built.

Common reversible HDL statements such as <=> (*swap*) or skip do not require special consideration as they already can be realized without additional circuit lines. For the remaining statements, including all reversible assignment operations with arbitrary combinations of binary operations, conditional statements, loops, and sub-modules, the proposed synthesis methods are described.

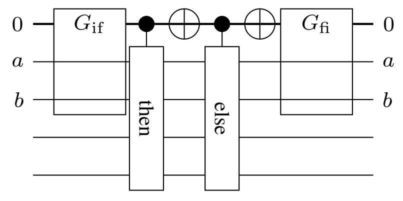
Reversible Assignment Statements: In order to realize statements of the form a⊕=e with e being an arbitrary expression composed of various binary operations, basically the respective building blocks are orchestrated as already illustrated in Fig. 4. First, a sub-circuit realizing the expression e, i.e. 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 reversible 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 4: Fig. 5 shows the circuit obtained by synthesizing cˆ=(a+b) using the proposed approach. The respective sub-circuits G_{a+b}, $G_{c\hat{}=a+b}$, and G^{-1}_{a+b} are highlighted by dashed rectangles. As can be seen, G^{-1}_{a+b} is obtained by reversing the order of the gates of G_{a+b}.

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



(a) Simplified conditional statement



(b) General conditional statement

Fig. 6. Synthesizing conditional statements

Conditional Statements: In order to realize conditional statements, a sub-circuit G_{if} evaluating the respective *if-expression* is created. The intermediate results of the respective expression are handled analogously to assignment statements as described above. Furthermore, an additional circuit line is applied to store the Boolean result of that expression. Then, control connections are applied to activate either the *then-* or the *else-block*. More precisely, control lines are added to all gates in the realization of the respective block. As a result, the gates in these blocks are triggered if, and only if, the result of the *if-expression* evaluates to 1 or 0, respectively. A NOT gate (i.e. a Toffoli gate without control lines) is thereby applied to flip the value of the additional line so that the gates of the *else-block* can be controlled as well. This flip is later restored by another NOT gate. Afterwards, the original constant value of the additional line is restored by applying the first sub-circuit G_{if} again.

Example 5: Fig. 6(a) illustrates the procedure assuming that the *if-expression* depends on a and b.

While this procedure can be applied in most of the cases (among others, also for the HDL code provided in Fig. 2), a problem arises if the *if-expression* depends on values which are modified within the *then-* or *else-block*. Then, the value of the additional line cannot be reversed by applying G_{if}, but a realization of the corresponding *fi-expression*.

Example 6: Consider the following HDL code:

```
if (a = b) then
  a += 2
else
  b += 2
fi (a = (b+2))
```

Here, the *if-expression* depends on values a and b which are modified in the *then-* and in the *else-block*, respectively. In order to ensure reversibility also in such cases, reversible description languages require the definition of a corresponding *fi-expression*. To realize a circuit for such an HDL description, the same approach as described above is applied, except for the last sub-circuit. Here, instead of G_{if} a newly generated sub-circuit realizing G_{fi} is applied. Fig. 6(b) illustrates this procedure. Now the additional line is not reversed by applying G_{if} but by applying G_{fi}.

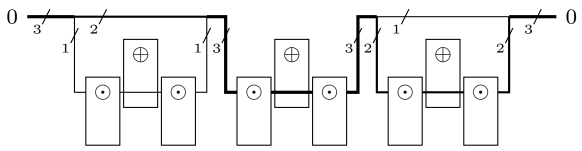


Fig. 7. Effect of expression size

Note that this procedure is recursively applied for nested conditional statements. Accordingly, the number of required additional lines grows linearly with the depth of these nested statements.

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.

V. Discussion

As the experiments in the next section confirm, applying the approach presented above leads to circuits with a significantly smaller number of circuit lines. In fact, every statement is synthesized with zero garbage outputs. Consequently, the number of additional lines can be determined by the statement that requires the largest number of additional lines in order to buffer intermediate results.

Example 7: Consider a sequence of three 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 respective circuit. Fig. 7 illustrates how these circuit lines are applied. For comparison, existing HDL synthesizers need $1+3+2 = 6$ additional circuit lines.

The number of additional circuit lines can be reduced further in many cases by restructuring the HDL 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 8: 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 described 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 circuit lines are needed to buffer the intermediate results.

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 number of lines usually is seen as the more important metric. Additionally, the increase in the gate costs is bounded. For example, in comparison to existing HDL synthesizers where the building blocks $G_\odot$ and $G_\oplus$ are applied for each assignment statement, the method proposed here 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. In fact, as the experiments in the next section show, the actual increase in the gate costs is often significantly smaller than this upper bound.

VI. Experimental Evaluation

We implemented an HDL synthesizer in C++ following the concepts introduced in Section IV and the optimizations discussed in Section V. Afterwards, we evaluated the approach using a broad variety of HDL descriptions provided in the SyReC language including components of a 32-bit processor, counters, or circuits performing arithmetic computations. Finally, we compared the obtained results to circuits generated

- with the HDL synthesizer presented in [12] and
- with the HDL synthesizer presented in [12] and additionally optimized with respect to the number of lines using the method presented in [15] and available in RevKit [21].

All evaluations have been conducted using a 64-bit Intel machine with 2.66 GHz and 4 GB of memory running Linux.

Table I summarizes all results obtained. The first column gives the name of the respective benchmarks, i.e. HDL descriptions. Afterwards, for each approach the total number of circuit lines (*lines*), the respective amount of additional circuit lines (*a.l.*), the number of gates (*gates*), the total gate costs[2] (*costs*), and the run-time (in CPU seconds) required to obtain the results are reported for each considered evaluation. Finally, percentage differences of the numbers for additional lines and costs are given in the last columns. More precisely, the results obtained by the HDL synthesizer from [12] are compared to (1) the results obtained by additionally applying [15] (denoted by [12] vs. [12]+[15]) and (2) to the results obtained by the proposed approach (denoted by [12] vs. proposed).

Based on the results, several interesting conclusions can be drawn. Applying post-synthesis optimization helps in some cases (e.g. *cond1*, *call2*), but has no effect for the majority of considered benchmarks. This can be explained by the nature of the applied optimization paradigm. In [15], a window-based re-synthesis scheme is used in order reduce the number of lines. The applicability of this approach is limited by the size of the respective windows. Since circuits obtained by HDL descriptions usually are composed of building blocks larger than the applicable window size, optimizations can rarely be obtained. Furthermore, the re-synthesis requires much more run-time and sometimes even exceeds the applied timeout of 1000 CPU seconds (denoted by *TO*).

[2]We only considered the established quantum cost metric (according to the table from [21]) in our evaluation.

TABLE I
EXPERIMENTAL RESULTS

Benchmark	Previously introduced approaches										Proposed Approach					Difference (in %)			
	HDL synthesizer from [12]					[12] + Line Reduction [15]										[12]vs.[12]+[15]		[12]vs. proposed	
	lines	a.l.	gates	costs	time	lines	a.l.	gates	costs	time	lines	a.l.	gates	costs	time	a.l.	costs	a.l.	costs
cpu_alu	5311	5107	20851	2363160	7.14	5311	5107	20851	2363160	TO	255	51	42169	4692080	0.85	0.00	0.00	99.00	-98.55
for	704	672	4536	9744	0.13	704	672	4536	9744	TO	64	32	4546	9754	0.02	0.00	0.00	95.24	-0.10
fibonacci2	543	444	7625	53661	0.10	543	444	7625	53661	TO	132	33	9236	64284	0.04	0.00	0.00	92.57	-19.80
plus10	160	128	864	1856	0.01	160	128	864	1856	8.75	64	32	866	1858	0.01	0.00	0.00	75.00	-0.11
alu2	235	137	4227	152852	0.06	233	135	4235	152953	53.03	133	35	8328	303474	0.06	1.46	-0.07	74.45	-98.54
simple_alu	235	137	15764	1851487	0.23	233	135	15772	1851588	253.01	133	35	31402	3700744	0.36	1.46	-0.01	74.45	-99.88
cond1	169	103	295	811	0.01	99	33	341	3539	10.20	99	33	440	1306	0.01	67.96	-336.37	67.96	-61.04
logic unit	203	105	385	6562	0.02	201	103	393	6663	3.03	133	35	612	10478	0.01	1.90	-1.54	66.67	-59.68
call2	160	64	496	992	0.01	128	32	465	961	1.03	128	32	496	992	0.01	50.00	3.13	50.00	0.00
call1	224	64	560	6066	0.01	224	64	560	6066	1.29	192	32	812	6566	0.02	0.00	0.00	50.00	-8.24
varops	288	192	1432	2928	0.02	276	180	1703	9766	45.78	192	96	2120	4368	0.02	6.25	-233.54	50.00	-49.18
cond2	69	3	1001	15113	0.01	69	3	1001	15113	0.01	68	2	1004	15124	0.01	0.00	0.00	33.33	-0.07
cond3	37	3	137	37817	0.01	37	3	137	37817	0.01	36	2	140	38524	0.01	0.00	0.00	33.33	-1.87
cpu_pc	134	36	326	4632	0.01	133	35	325	4639	0.24	132	34	338	4656	0.01	2.78	-0.15	5.56	-0.52
															Average:	6.94	-29.92	61.97	-35.54

Legend of columns:
lines: total number of circuit lines *a.l.*: respective amount of additional circuit lines *gates*: number of gates *time*: required run-time
[12] *vs.* [12]+[15]: percentage differences between the results obtained by [12] and the results obtained by additionally applying [15]
[12] *vs. proposed*: percentage differences between the results obtained by [12] and the results obtained by the proposed approach

In contrast, the proposed approach enables reductions in the number of additional lines for all cases. Most of the achieved reductions are considerably large. In the best case (*cpu_alu*), the number of additional circuit lines can be reduced from 5107 to 51, i.e. by two orders of magnitude. On average, improvements by 60% are observed.

As discussed in Section V, these achievements come at the expense of larger gate costs. However, the increase in the gate costs is acceptable considering that usually the number of lines is seen as the more important metric.

Finally, the proposed improvements do not affect the run-time efficiency of the actual synthesis. The HDL synthesizer still generates all results in almost no time.

VII. CONCLUSION

In this work, we proposed an HDL synthesizer generating reversible circuits with a significantly smaller amount of additional lines. Intermediate results from binary operations are reversed as soon as they are not required anymore. Additional gates are used for this purpose. As confirmed by experiments, our solution leads to reversible circuits with 60% less additional lines on average at the expense of an acceptable increase of the total gate costs.

Using the proposed approach, now the number of additional circuit lines solely depends on the statement that requires the largest number of additional lines to buffer intermediate results. As discussed in Section V, this amount can further be decreased by restructuring HDL statements. Besides that, these lines can only be removed if corresponding building blocks for the respective ream of possible combinations between reversible assignment operations and binary operations can be provided. As a result, the proposed approach presents a basis for further research which is left for future work.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) (DR 287/20-1).

REFERENCES

[1] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
[2] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," *Foundations of Computer Science*, pp. 124–134, 1994.
[3] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff, "Limits to binary logic switch scaling – a gedanken model," *Proc. of the IEEE*, vol. 91, no. 11, pp. 1934–1939, 2003.
[4] 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*, vol. 414, p. 883, 2001.
[5] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz, "Experimental verification of Landauer's principle linking information and thermodynamics," *Nature*, vol. 483, pp. 187–189, 2012.
[6] 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*, 2012, pp. 1036–1041.
[7] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 22, no. 6, pp. 710–722, 2003.
[8] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conf.*, 2003, pp. 318–323.
[9] P. Gupta, A. Agrawal, and N. K. Jha, "An algorithm for synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 25, no. 11, pp. 2317–2330, 2006.
[10] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Design Automation Conf.*, 2009, pp. 270–275.
[11] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler, "Synthesis of reversible circuits with minimal lines for large functions," in *ASP Design Automation Conf.*, 2012, pp. 85–92.
[12] R. Wille, S. Offermann, and R. Drechsler, "SyReC: A programming language for synthesis of reversible circuits," in *Forum on Specification and Design Languages*, 2010, pp. 184–189.
[13] T. Yokoyama and R. Glück, "A reversible programming language and its invertible self-interpreter," in *Symp. on Partial evaluation and semantics-based program manipulation*, 2007, pp. 144–153.
[14] H. B. Axelsen, "Clean translation of an imperative reversible programming language," in *Int'l Conf. on Compiler Construction*, 2011, pp. 144–163.
[15] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Design Automation Conf.*, 2010, pp. 647–652.
[16] D. Maslov and G. W. Dueck, "Reversible cascades with minimal garbage," *IEEE Trans. on CAD*, vol. 23, no. 11, pp. 1497–1509, 2004.
[17] R. Wille, O. Keszöcze, and R. Drechsler, "Determining the minimal number of lines for large reversible circuits," in *Design, Automation and Test in Europe*, 2011, pp. 1204–1207.
[18] 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*, 2008, pp. 220–225, RevLib is available at http://www.revlib.org.
[19] Y. Takahashi and N. Kunihiro, "A linear-size quantum circuit for addition with no ancillary qubits," *Quantum Information and Computation*, vol. 5, pp. 440–448, 2005.
[20] S. Offermann, R. Wille, G. W. Dueck, and R. Drechsler, "Synthesizing multiplier in reversible logic," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010, pp. 335–340.
[21] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: An Open Source Toolkit for the Design of Reversible Circuits," in *Reversible Computation 2011*, ser. Lecture Notes in Computer Science, vol. 7165, 2012, pp. 64–76, RevKit is available at www.revkit.org.