Towards VHDL-based Design of Reversible Circuits

Work in Progress Report

Zaid Al-Wardi[1,2], Robert Wille[3,4], and Rolf Drechsler[1,4]

[1]Institute of Computer Science, University of Bremen, D-28359 Bremen, Germany
[2]Collage of Engineering, Al-Mustansiriya University, Baghdad, Iraq
[3]Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
[4]Cyber-Physical Systems, DFKI GmbH, D-28359 Bremen, Germany
{alwardi,drechsle}@informatik.uni-bremen.de robert.wille@jku.at

Abstract. *Hardware Description Languages* (HDL) facilitate the design of complex circuits and allow for scalable synthesis. While rather established for conventional circuits, HDLs for reversible circuits are in their infancy and usually require a deep understanding of the reversible computing concepts. This motivates the question whether reversible circuits can also efficiently be designed with conventional HDLs, such as VHDL. This work discusses this question. By this, it provides the basis towards a design flow that requires no or only little knowledge of the reversible computation paradigm which could ease the acceptance of this non-conventional computation paradigm amongst designers and stakeholders.

1 Introduction

The majority of reversible circuit design and synthesis methodologies are derived from functional descriptions provided in terms of truth tables, two-level descriptions, decision diagrams, or similar (Boolean) function representations (see e.g. surveys provided in [1, 2]). These approaches are limited by their restricted scalability and are not competitive to the state-of-the-art design flows available for conventional circuits.

Hardware Description Languages (HDL) address scalable design of digital circuits [3]. In fact, the design of conventional circuitry heavily relies on established HDLs such as VHDL or Verilog. For reversible circuit design, a clear trend towards higher levels of abstractions can be seen [4, 5]. The proposed approaches employ the reversible computation paradigm with its characteristics as well as restrictions and, hence, rely on dedicated concepts such as reversible assignments, reversible control logic, etc. Since, historically, design focused on circuits following the conventional computing paradigm, those concepts are usually rather unfamiliar amongst HDL-designers.

This motivates the question whether reversible circuits can also efficiently be designed with conventional HDLs such as VHDL or Verilog. Obviously, this would break with many concepts and may lead to drawbacks such as the need to

```
1      entity test is
2          port (a,b: in bit; f: out bit);
3      end entity test;
4
5      architecture dataflow of example is
6          signal w: bit;
7      begin
8  S1:     f <=    a and w;
9  S2:     w <=    not b;
10     end architecture dataflow;
```

Fig. 1. Simple VHDL program

embed non-reversible HDL description means into reversible circuitry (causing overhead e.g. in terms of additional circuit lines).

In this work, we address this issue and choose the widely used hardware description language VHDL as an example of a conventional HDL. We discuss its suitability to synthesize reversible circuits. The findings from the resulting observations provide the basis towards a design flow that requires no or only little knowledge of the reversible computation paradigm. At the same time, it pinpoints to the weaknesses and open issues to be addressed in order to make VHDL-based design indeed a more accessible alternative to the existing design solutions for reversible circuits. Possible directions how to address these weaknesses are discussed in this work.

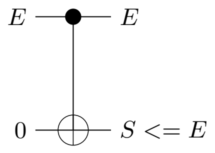
2 Realizing VHDL Signals

VHDL signal types can directly be mapped to signals of the reversible circuits. More precisely, a VHDL signal is mapped to a reversible circuit line[1].

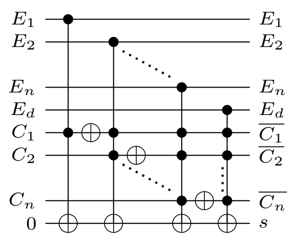
In Fig. 1 we can see a VHDL code that declares different types of signals, which are mapped to lines with different specifications as follows:

1. **Input ports** `a,b` : These lines carry input values to the circuit and remain unchanged within a circuit.
2. **Output port** `f` : This has a constant $'0'$ input, then an expression is assigned to this signal (line) by a statement within the architecture body.
3. **Internal signal** `w` : This line represent an internal wire. It is similar to output ports in that it is initially constant $'0'$ and assigned in the same way as well. The difference between outputs and wires is that wires facilitate computing other signal(s) and then are considered garbage outputs.
4. **Implicit lines**: These lines are similar to internal signals in that they have constant $'0'$ inputs and constitute garbage outputs, but are not explicitly declared within the code. Such lines are mandatory to compute non-reversible operations, e.g. to compute the expression (`a and w`) in Fig. 1, line(8).

[1] For simplicity, in the following a line refers to an N-line bundle representing an N-bit signal (accordingly, a single line in figures represent an N-bit circuit line-bundle).



a. Simple assignment



b. Conditional signal assignement

Fig. 2. Realization of signal assignment

3 Realizing VHDL statements

With the signals defined and initially realized in the circuit, the realizations of the respective operations in terms of reversible gates can be conducted. To this end, all statements in the VHDL code are traversed and synthesized. A statement is considered as a sub-system that performs some action to realize the desired operation.

3.1 Signal Assignment

A statement, in its simplest form, is usually composed of an expression which is evaluated and whose result is afterwards assigned to a circuit signal (i.e. a statement usually has the form (`S <= E;`), with `E` being the expression and `S` being the signal to which the result is assigned). The realization of the underlying expressions is covered afterwards in the following section. Realizing signal assignment (a non-reversible operation) is possible when the target signal is known to be a constant $'0'$ [6]. This assignment is realized using Toffoli gates, as shown in Fig. 2.a.

Conditional signal assignment statements appear in the following form: (`S <= E1 `**`when`**` C1 `**`else`**` E2 `**`when`**` C2 ... `**`else`**` En `**`when`**` Cn `**`else`**` Ed;`). This assignment requires a case distinction to decide which expression is to be assigned to the target signal. Fig. 2.b shows a possible realization for this.

3.2 Components

Components are entities instantiated within the architecture of another entity. Each instance places a sub-circuit definition within the main circuit. Fig. 3 shows a VHDL code that declares a component, then instantiates it twice within the architecture body.

This structural style of describing systems is preferred for synthesis purposes. Component sub-circuits should be determined first, and this sub-circuit definition is to be placed in the main circuit for each instant. The only change is the mapping of component lines into the main circuit lines; therefore a `port map` is associated with each instance to serve as a look-up table for this mapping, as shown in Fig. 4.

```
    entity main is
        port(   x,y,z: in bit; result: out bit);
    end entity main;

    architecture structural of main is
        component test is
            port(a,b: in bit; f: out bit);
        end component test;
        signal temp: bit;
    begin
L1:     test port map (a => x, b => y, f => temp);
L2:     test port map (a => temp, b => z, f => result);
    end architecture structural;
```

Fig. 3. Structural VHDL architecture with declared and instantiated components

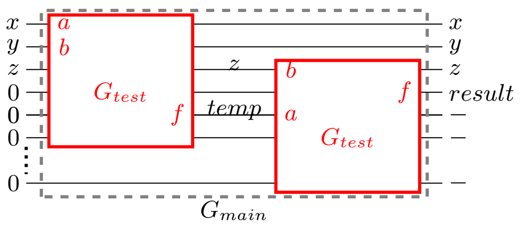


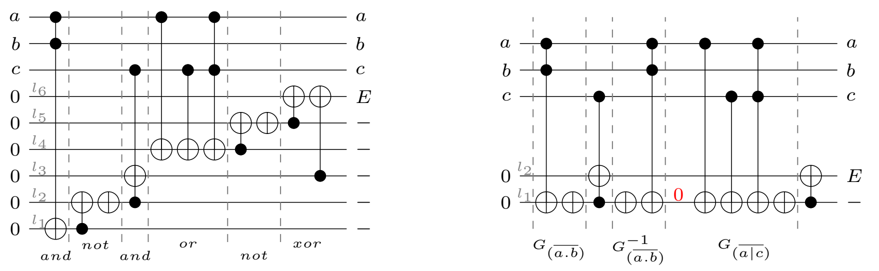
Fig. 4. Using component circuits to synthesize the VHDL code from Fig. 3

4 Realizing Expressions

Up to this point, the discussion assumes that expressions are values that are, somehow, available on certain circuit lines like any other signal. This skips a core issue, namely how to realize expressions. VHDL provides a set of operations to be used in expressions. These operations are not necessarily reversible. An additional line with constant inputs is applied to make a non-reversible function reversible [7] (leading to the implicit lines as discussed in Section 2). This is exactly how the reversible HDL SyReC tackles this problem [4]. Hence, realizing an expression E which is combined with N operators will implicitly add N constant lines to the circuit. This is considered a serious drawback [8].

Line-awareness when realizing HDL expressions can tangibly increase the overall efficiency of this approach [9]. The reduction can be started by reconsidering the necessity of adding lines in some special cases, such as with `not`, `xor`, `+` and `-`. These operators are reversible, hence, can be computed with no additional line. Further reduction in lines may be obtained by reverse computing (re-computing) intermediate values and reusing these lines for further computations [9]. To reverse a computation, just repeat it in the reverse order of gates.

Example 1. Consider the Boolean expression E $(\overline{a.b}.c \oplus \overline{a|c})$, which has the following form in VHDL: (**not**(a **and** b) **and** c **xor** **not**(a **or** c)). The value of E is computed based on six Boolean operations. Hence, six constant input lines are required to compute this expression. Fig. 5.a shows a reversible circuit



a. Direct realization of expression E b. Line-aware realization of expression E

Fig. 5. Circuits realizing expression E from Example 1

to compute E. Fig. 5.b shows the line-aware realization of the same expression using only two constant lines, in which $G^{-1}_{\overline{(a.b)}}$ re-computes line l_1. This line is used once more to compute the sub-expression (`not (a or c)`), using $G_{\overline{(a|c)}}$.

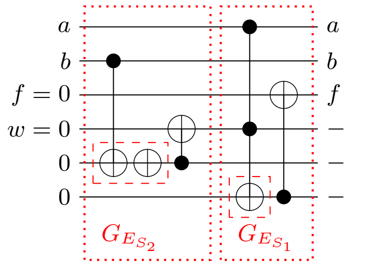
5 Overall Realization

Using the realization schemes described above for signals, statements, and expressions, an overall realization can be obtained for a given VHDL code. To this end, the respectively obtained sub-circuits need to be accordingly connected. In conventional hardware, it does not matter which statement is synthesized first, the resulting hardware will be exactly the same because of statements' concurrency [3]. The reversible computation scheme, on the other hand, is processing signals in a cascade fashion. Consequently, signals are successively computed. A simple algorithm, based on signal dependence, can be applied to determine the correct order in which statements are to be synthesized. Hence, the order in which statements are synthesized may differ from the order in which they appear in the code. Fig. 6.a shows such an example. The figure shows the correct realization of the VHDL code from Fig. 1, where statement `S2` is synthesized before `S1` to resolve the issue of signal dependence.

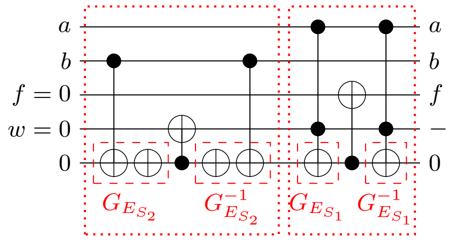
The two statements `S1` and `S2` from Fig. 1 have expressions on their right hand sides. A constant $'0'$ line is needed, in this example, to compute each expression. As a result, two implicit lines are added to realize the circuit (see Fig. 6.a). For complex codes, implicit lines keep accumulating throughout the code – resulting in large numbers of circuit lines. A line-aware realization on the overall module level may also re-compute lines to realize garbage-free statements [4]. This allows statements to reuse implicit lines. In Fig. 6.b, the implicit line used for statement `S2` is re-computed and then reused for `S1`. This arrangement realizes the circuit with only one implicit line – compared to the two lines needed in the circuit shown in Fig. 6.a.

6 Conclusions

In this work, we discussed how to realize VHDL code as reversible circuits. To this end, we considered the realization of the corresponding signal declarations,



a. Direct realization



b. Line-aware realization

Fig. 6. Circuits realization of VHDL code from Fig. 1

statements, as well as expressions. Based on that, two different schemes for the overall realization of the desired circuit have been proposed – with a particular focus on the number of eventually resulting circuit lines. With these contributions, we provide an initial basis towards a VHDL-based reversible circuit design flow that requires no or only little knowledge of the reversible computation paradigm. For future work, it is planed to consider more data-types with associated operators, as well as covering more statements and settings.

Acknowledgments

This work has partially been supported by the European Union through the COST Action IC1405.

References

1. Drechsler, R., Wille, R.: From truth tables to programming languages: Progress in the design of reversible circuits. In: Int'l Symp. on Multi-Valued Logic. (2011) 78–85
2. Saeedi, M., Markov, I.L.: Synthesis and optimization of reversible circuits - a survey. ACM Computing Surveys (2011)
3. Ashenden, P.J.: The Designers Guide to VHDL. 3 edn. Elsevier (2008)
4. Wille, R., Schönborn, E., Soeken, M., Drechsler, R.: SyReC: A hardware description language for the specification and synthesis of reversible circuits. INTEGRATION, the VLSI Jour. **53** (2016) 39–53
5. Thomsen, M.K.: A functional language for describing reversible logic. In: Forum on Specification and Design Languages. (2012) 135–142
6. Wille, R., Soeken, M., Drechsler, R.: Reducing the number of lines in reversible circuits. In: Design Automation Conf. (2010) 647–652
7. Wille, R., Keszöcze, O., Drechsler, R.: Determining the minimal number of lines for large reversible circuits. In: Design, Automation and Test in Europe. (2011)
8. Wille, R., Soeken, M., Miller, D.M., Drechsler, R.: Trading off circuit lines and gate costs in the synthesis of reversible logic. INTEGRATION, the VLSI Jour. **47**(2) (2014) 284–294
9. Alwardi, Z., Wille, R., Drechsler, R.: Towards line-aware realizations of expressions for HDL-based synthesis of reversible circuits. In: Reversible Computation. (2015) 233–247