Synthesis of Reversible Circuits Using Conventional Hardware Description Languages

Zaid Alwardi*† Robert Wille‡§ Rolf Drechsler*§

*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
†Collage of Engineering, Al-Mustansiriya University, Baghdad, Iraq
‡Institute for Integrated Circuits, Johannes Kepler University Linz, A-4040 Linz, Austria
§Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{alwardi,drechsler}@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, HDL-based design of reversible circuits is in its infancy. This motivates the question whether conventional HDLs can also be efficiently used for the design of reversible circuits. This work investigates this question and provides a basis towards a design flow that requires only little knowledge of reversible computation. This eases the acceptance of this non-conventional paradigm amongst designers and stakeholders.**

I. Introduction

The reversible computing paradigm is receiving increasing attention (in particular for so-called emerging technologies) and provides the basis for several applications including but not limited to *quantum computation* [1], certain aspects of *low-power design* [2], the design of *adiabatic circuits* [3], [4], interconnects [5], [6], *encoding and decoding* devices [7], or *verification* [8]. Reversible circuits, by definition, can only realize bijective operations, i.e., functions that map each possible input vector to a *unique* output vector. Then, computations can be made in either direction. Due to this property, many special characteristics and restrictions apply for reversible circuits, which makes them rely on a significantly different computation paradigm. Already a simple standard operation like the logical AND illustrates the differences, as it is not possible to determine the input values when the AND output is 0.

As a consequence, new methods for the design and synthesis of reversible circuits have been introduced. Thus far, the majority of them focused on the realization of reversible circuits derived from functional descriptions provided in terms of truth tables [9], [10], two-level descriptions [11], decision diagrams [12], [13], [14], or similar (Boolean) function representations. Obviously, these approaches are limited by their restricted scalability and are not competitive to the state-of-the-art design flows available for conventional circuits and systems.

Hardware Description Languages (HDL) are an obvious direction to address this problem. In fact, the design of conventional circuitry heavily relies on established HDLs such as VHDL or Verilog. For reversible circuit design, there are hardly any comparable design languages available. However, a clear trend towards higher levels of abstractions can be seen and reversible description languages have recently been proposed [15], [16]. They employ dedicated reversible computations with their concepts and constrains, such as, reversible signal assignments and reversible control logic. But since, historically, the 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 be designed with conventional HDLs such as VHDL. Obviously, this would break with many concepts and may lead to drawbacks such as the need to embed non-reversible HDL description means into reversible circuitry (causing overhead e.g., in terms of additional circuit lines). However, a more accessible HDL and, hence, design flow may compensate for this. Unfortunately, rather few discussions and evaluations on the "costs" of designing reversible circuits using conventional HDLs have been conducted yet.

In this work, we investigate reversible circuit synthesis from the widely used hardware description language VHDL. The findings from the resulting observations provide the basis towards a design flow that requires only little knowledge of the reversible computation paradigm.

The remainder of this work is structured as follows: Section II provides a brief review of VHDL and the basics of reversible circuits. Afterwards, the realization of signals declared in the VHDL code is discussed Section III. Then, the realizations of VHDL expressions and signal assignments are covered in Section IV. The overall interconnection of statements and components are discussed in Section V. Improving the resulting circuits is proposed in Section VI. The findings are discussed with case studies in Section VII, by which differences and similarities of the proposed approach is analyzed in comparison to a dedicated reversible HDL (namely SyReC proposed in [15]). Finally, the paper is concluded in Section VIII.

II. Preliminaries

This section provides the necessary background to keep the paper self-contained. It includes a brief review of VHDL that is commonly used to describe circuits as well as a brief review of reversible circuits.

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

Fig. 1. Behavioral VHDL description

```
1  entity main is
2     port( q,r,s: in bit; y: out bit);
3  end entity main;
4
5  architecture structural of main is
6     component sub is
7          port(a,b: in bit; f: out bit);
8     end component sub;
9     signal t: bit;
10  begin
11 L1:      test port map (a => q, b => r, f => t);
12 L2:      test port map (a => t, b => s, f => y);
13 end architecture structural;
```

Fig. 2. Structural VHDL description

A. Introducing VHDL

VHDL is a hardware description language designed to allow the description of the structure of a circuit, i.e., its decomposition into subsystems as well as their interconnections. To this end, established programming styles and different levels of abstractions are utilized. Using VHDL, circuits can be simulated, synthesized and verified before being manufactured [17]. More precisely, a circuit is first defined by an *entity declaration*, which introduces a name for the entity and lists the ports (input and output signals). Hence, an entity declaration describes only the external view of the design.

The internal implementation of an entity is provided in an *architecture body* of that entity. Architectures might be provided in different fashions: A *behavioral* architecture body describes the function in an abstract way in terms of *process statements*. A process statement defines a sequence of operations that are to be executed when the circuit is simulated. To this end, a wide variety of actions might be included within a process statement, which (in some cases) restricts the synthesizebility of the architecture.

Synthesis-oriented designers prefer an alternative model to describe architecture implementations of entities, which is called *structural* description. This model describes the circuit in terms of a net-list of sub-circuits. More precisely, sub-circuits are declared as *components*. A number of *component instances* (i.e., copies) may appear in the architecture body to represent these subsystems. A component instance includes a *port map* to specify the interconnections of these component instances within the enclosed architecture body.

Another possible description is by *signal assignment* statements, which define the flow of data to compute signals. An architecture body completely described using signal assignment statements is usually referred to as a *data-flow* description style. Often it is useful to describe the required system using a mixture of processes, interconnected components, and signal assignment statements.

In the remainder of this paper, we focus on the main descriptions provided by VHDL for the purpose of reversible circuit synthesis. Simulation related description details, including process statement actions, are not covered. Also, circuits that contain feedback are not supported, because feedback connections are not directly allowed in the reversible circuit paradigm.

Example 1. *Fig. 1 provides an example of a data-flow VHDL circuit. The entity* `sub` *has three single-bit ports, namely two input ports (*`a,b`*), and one output port (*`f`*). A single-bit wire signal (*`x`*) is declared within the architecture body. The implementation of this system contains two signal assignment statements. The first statement (S1) computes the wire signal* `x`*, while the second statement (S2) computes the output signal* `f`*. Fig. 2, on the other hand, shows a structural description of a VHDL architecture (*`main`*), in which the entity* `sub`*, defined in Fig. 1, is declared as a component and then instantiated twice (statements* `L1` *and* `L2`*). The* `port map` *associated with each instance defines the inter-connectivity of this specific component-instance within the* `main` *circuit.*

B. Reversible Functions and 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 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.

In this paper, reversible circuits are realized by *Toffoli gates*. A Toffoli gate uniquely maps the input set of signals ($\mathbf{X} = \{x_1, x_2, \dots, x_j, \dots, x_n\}$) to the output ($\mathbf{X}' = \{x_1, x_2, \dots, x_{i_1} x_{i_2} \cdots x_{i_k} \oplus x_j, \dots, x_n\}$). That is, a Toffoli gate inverts the *target line* x_j if, and only if, all *control lines* are assigned the logic value 1.

By definition, reversible circuits can only realize reversible functions. To realize non-reversible functions, *additional circuit lines* with constant inputs and garbage outputs are applied (see e.g., [18]) – yielding an *embedding*. Furthermore, additional circuit lines are also used frequently in hierarchical synthesis approaches (see e.g., [12], [15]).

Example 2. *Fig. 3 shows a reversible circuit formed by cascading Toffoli gates g_1 and g_2. The gate g_1 shows an example of embedding a non-reversible operation AND within a reversible circuit by using a constant-input line. On the other hand, g_2 shows that a reversible XOR operation does not need such embedding.*

III. VHDL Signals in Reversible Circuits

VHDL signals are represented as nodes in a conventional circuit, where the value of this signal can be measured. Circuit components should be properly interconnected to compute the desired signals to drive these nodes.

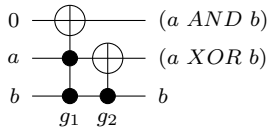


Fig. 3. Reversible gates realizing Boolean operations

VHDL signal types can be mapped directly 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. These signals are mapped to circuit lines with different specifications. **Input ports** (`a,b`) carry input values to the circuit. Such lines remain unchanged within the circuit. In other words, these signals do not appear in the left hand side of an assignment statement. **Output ports** (`f`) are constant $'0'$ inputs to which the output of an expression is assigned to (realizing a statement within the architecture body). **Internal wires** (`x`) are similar to output ports as they have constant $'0'$ inputs and are assigned in the same way as well. The difference between outputs and wires is that they are valid only inside the architecture body to facilitate internal computations and, afterwards, are considered garbage outputs.

The fact that VHDL operations are not necessarily reversible leads to inevitable **implicit lines** with constant inputs that are not explicitly associated with signals but are required to compute expressions (e.g., `(a and x)` in Fig. 1, statement `S2`).

Up to this point, the scheme of a reversible circuit realizing a VHDL code is composed of empty lines only without any gates. In other words, a circuit that computes nothing. Gates are added to process signals on circuit lines to compute the desired outputs as described by the statements in the architecture body.

IV. Signal Assignment Statements

A simple signal assignment (`S <= E`) is composed of three parts: a target signal `S`, an assignment operator <=, and a right-hand-side expression `E`. The statement is realized in two steps: The first is to compute the RHS expression E, i.e., realize the circuit G_E. Then, the second step is to assign the computed value `E` to the target-signal `S`.

A. Computing Expressions

VHDL provides a set of operations, such as Boolean, arithmetic, comparison, and so on. The operations are applied on operand signals to compose VHDL expressions. These operations are not necessarily reversible. Hence, an additional line with constant inputs is applied to make a non-reversible function reversible [18] – leading to the implicit lines discussed in Section III. This is exactly how the reversible HDL SyReC tackles this problem [15]. 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 [20] (to be discussed in Section VI-A).

[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).

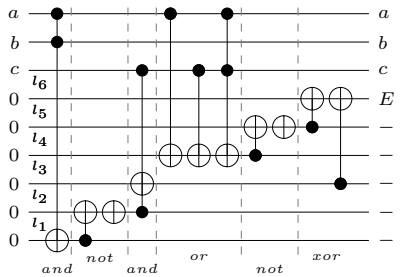


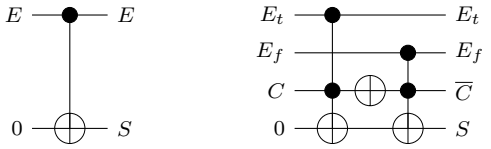
Fig. 4. Circuits realizing expression E from Example 3

Example 3. *Fig. 4 shows a reversible circuit to compute the VHDL expression* `E`*, which is given by* (`not(a and b) and c xor not(a or c)`)*. The expression is computed based on six Boolean operations. Hence, six constant input lines are applied to the circuit.*

B. Assignment Operation

The signal assignment operation is irreversible because it leads to a loss of the initial value of the target signal. Consequently, a circuit line with a constant input is inevitable to realize such an operation. A Toffoli gate can be used to copy the value of a line `E` into line `S`, if and only if, `S` is a constant $'0'$, as shown in Fig. 5(a), because (`(E xor 0)=E`). The operation is a simple assignment (`S <= E`), by which the expression `E` (covered in Section IV-A) is assigned to the target signal `S`, which is known to be a constant $'0'$ [2] (see Section III).

Conditional signal assignments are also provided in VHDL, with the form (`S <= Et when C else Ef`), by which `Et` is assigned to the target signal `S` only when the condition `C` is evaluated to $'true'$ or $'1'$, while `Ef` is assigned otherwise, as shown in Fig. 5(b). Here, the three expressions (`Et, Ef and C`) have to be computed before applying the assignment operation. A conditional assignment may be extended to multiple-conditionals (e.g., see Fig. 10(b)).



(a) Simple (b) Conditional

Fig. 5. Realization of signal assignment

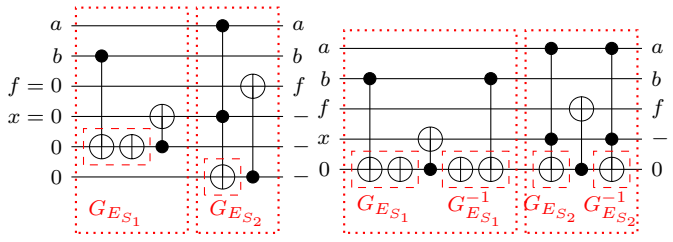
V. Interconnecting Sub-Circuits

An overall circuit realization for a given VHDL code is computed by interconnecting sub-circuits of all statements together within one main circuit. This includes all expressions, assignments, and instances of components.

A. Statement Cascade

A fundamental difference between conventional and reversible paradigms must be addressed here. In conventional circuits, no matter which statement you synthesize first, the

[2]A target-signal can only be an output or a wire. Multiple assignments means driving a node from different circuit outputs in the conventional paradigm.



(a) Basic realization (b) Line-aware realization

Fig. 6. Reversible circuits realizing the VHDL code from Fig. 1

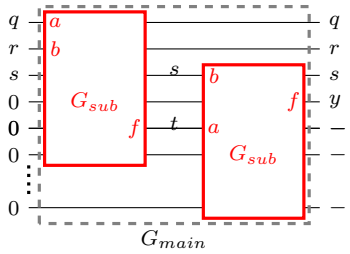


Fig. 7. Using component circuits to synthesize the VHDL code from Fig. 2

result would be the same hardware because of statement concurrency. The reversible computation paradigm, on the other hand, is successively processing signals by cascaded gates. Consequently, signals are successively (not concurrently) computed. In this regard, the order in which the statement are considered has an effect.

Example 4. *The VHDL code in Fig. 1 contains two assignment statements, each statement has an expression with one operator on its right hand side. Consequently, two implicit lines are expected. The resulting circuit is shown in Fig. 6(a).*

B. Components

As reviewed in Section II-A, the structural style describes systems as a set of interconnected components. Components are entities instantiated within the architecture of another entity. Each instance places a sub-circuit definition within the main circuit. Fig. 2 shows a VHDL code that declares a component and, then, instantiates it twice within the architecture body. A component sub-circuit should be determined first, then a copy of this sub-circuit is placed within the main circuit for each instance. The only difference is the mapping of component signals into the main circuit signals; therefore a `port map` is associated with each instance to serve as a look-up table for this mapping. This is illustrated in Fig. 7 that shows the component interconnection of the structural description from Fig. 2.

VI. Improving the Circuit Realization

Realizing expressions and other non-reversible actions implicitly adds constant lines to the circuit. These lines are accumulated throughout statements and result in circuits with a large number of constant inputs, which is the main drawback of all hierarchical approaches [20].

In this section, we propose some arrangements to reduce the number of lines and/or gate costs, without compromising the main advantage of this approach (scalability).

A. Line-aware Synthesis

According to the interconnection suggested in Section V, implicit lines are assigned and used only once within the architecture body – their outputs are garbage, i.e., not usable again in the circuit. In contrast, realizing a statement with no garbage is possible when the RHS expression is computed in the reverse direction (re-computed). This technique has been proposed for line-aware SyReC synthesis [21]. More precisely, in addition to the two steps from Section IV, a third step appends the inverse circuit G_E^{-1} to the circuit cascade to re-compute the garbage lines used to compute E back to constant 0. The next statements will then reuse the same lines. In this way, circuits can be realized with less lines.

Example 5. *Fig. 6(b) shows the reversible circuit realization for the VHDL code from Fig. 1 following this scheme. The circuit requires only* 1 *implicit line instead of* 2 *lines as in the circuit from Fig. 6(a).*

The re-compute technique proposed here works on the register transfer level and it trades-off lines with extra circuitry that almost doubles the circuit cost.

B. Gate Level Complexity Reduction

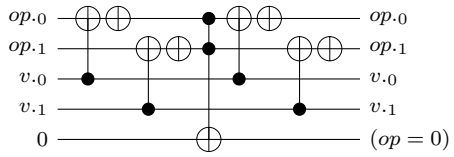
A constant-input is not a signal applied to the circuit, but it is more like a literal numeric value in the code. In the conventional realization of VHDL codes, numbers (i.e., literals) do not require circuits to compute their values as they are already given in the code. Furthermore, an operation on a number operand can dramatically reduce the complexity of the circuit.

In the reversible circuit paradigm, numbers are represented as constant inputs. The use of a constant-input for each number in the code reduces the quality of the circuit realization. On the other hand, considering constant lines, gate complexity can be reduced, e.g., (1) remove any control with a constant $'1'$ from the gate and (2) remove any Toffoli gate with one control line known to be constant $'0'$.

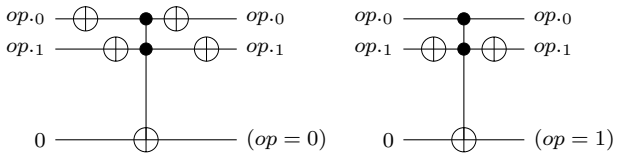
Example 6. *Fig. 8(a) shows the circuit of a 2-bit equality operation* `(op = v)`*, where* `op` *and* `v` *are both variables. In the VHDL code of Fig. 10(b), we can see a special case of this operation used as conditions, e.g.,* `(op = 0)`*. In this case, one of the operands is a constant number instead of a variable signal. Applying the complexity reduction rules as suggested above results in Fig. 8(b), which uses less lines and lower gate cost as well. Applying the same optimization on a condition with a different number, such as* `(op = 1)`*, results in a different circuit (namely the one shown in Fig. 8(c)).*

VII. Discussion

This section discusses the proposed VHDL-based synthesis. We consider two cases to introduce differences and similarities between the proposed approach and the reversible-specific HDL solution SyReC introduced in [15]. The cases are neither meant to evaluate, nor to decide a clear winner between VHDL and SyReC. Therefore they are chosen to be simple enough



(a) Variable signal $op = v$



(b) Constant number $op =' 0'$ (c) Constant number $op =' 1'$

Fig. 8. Gate level optimization of a constant input

```
entity gray2binary is
  port( g : in STD_LOGIC_VECTOR (3 downto 0);
        b : out STD_LOGIC_VECTOR (3 downto 0));
end entity gray2binary;

architecture Behavioral of gray2binary is
begin
  b(3) <= g(3);
  b(2) <= g(3) xor g(2);
  b(1) <= g(3) xor g(2) xor g(1);
  b(0) <= g(3) xor g(2) xor g(1) xor g(0);
end behavioral;
```

(a) Basic description

```
architecture Behavioral of gray2binary is
  signal w (2 downto 0);
begin
  w(2) <=  g(3) xor g(2);
  w(1) <=  w(2) xor g(1);
  w(0) <=  w(1) xor g(0);
  b(3) <= g(3);
  b(2) <= w(2);
  b(0) <= w(0);
end architecture behavioral;
```

(b) Optimized description

```
module gray2binary(inout x(4))
  x.2 ^= x.3
  x.1 ^= x.2
  x.0 ^= x.1
```

(c) SyReC description

Fig. 9. HDL descriptions of a 4-bit gray-code to binary converter

for explaining the impact of the respectively obtained synthesis from each approach on the resulting circuits.

Between both solutions, one fundamental difference is the definition of signal assignment, which is irreversible in VHDL (<=) and reversible in SyReC (^=, i.e., by additionally employing, e.g., an xor-assignment which however might require the addition of `out` and `wire` signals to realize the intended functionality). Computing expressions, conditionals, and components using constant inputs is similar in VHDL as compared to SyReC.

Metrics used to measure circuit complexity in the following cases are: (1) *Gate-count*; the total number of gates in the circuit. (2) *Lines*; the total number of lines used to compute the circuit. (3) *Quantum-cost*; the total sum of number of elementary quantum gates used to map the gates in the circuit, as defined in [22]. (4) *Transistor-cost*; estimates the effort needed to realize all reversible gates in the circuit using CMOS technology according to [23].

A. Case Study: Gray-code to Binary Code Conversion

Encoders and decoders are identified as typical reversible computations [7]. In the following, a 4-bit Gray-code to Binary-code converter is studied. Fig. 9(a) shows a VHDL description[3] of this converter. The code defines two 4-bit vectors: `(g)` for the input Gray-code and `(b)` for the output Binary-code. The architecture description in this code incorporates some repeated computations, e.g., `(g(3) xor g(2))` is computed three times. Hence, an equivalent description is shown in Fig. 9(b) to reduce the resulting computation complexity (and, by this, the circuit cost). This code declares a three-bit wire `w` to facilitate the computations. Despite being described using more statements, and explicitly declaring internal wires, this code is better realized as reversible circuits than the first code. This is confirmed by the number shown in Table I, in which the costs of each realization are given. Here, **(V1)** provides the values of the basic realization (see Sections IV and V) and **(V2)** the values of the improved realization (see Section VI).

We additionally consider a description provided in SyReC syntax as shown in Fig 9(c). This code conversion is reversible, because of the one-to-one correspondence between the two codes. Hence, the Gray-code to Binary-code converter is an ideal example to demonstrate the merits of SyReC-based synthesis (in its current state of development) compared to VHDL-based synthesis introduced above. The SyReC approach performs significantly better because reversibility can fully be exploited.

TABLE I
RESULTS FOR THE GRAY-CODE TO BINARY-CODE CONVERTER

Parameter	VHDL				SyReC
	Fig. 9(a)		Fig. 9(b)		Fig. 9(c)
	V1	V2	V1	V2	
Gates	16	28	10	16	**3**
Total lines	14	11	11	9	**4**
Quantum cost	16	28	10	16	**3**
Transistor cost	128	224	80	128	**24**

B. Case Study: Logic Unit

Unlike the first case, the second is an irreversible 32-bit logic unit. Here, Fig. 10(a) and Fig. 10(b) show two equivalent codes to describe this logic-unit in SyReC[4] and VHDL, respectively. The output signal `x0` is computed by a conditional assignment. In the SyReC code, `x0` is initialized using the xor-operator (^=), e.g., in `x0 ^= (x1 & x2)`. Here, the operation is identical to (<=) in VHDL, since `x0` is an `out` signal.

Table II shows the result of SyReC configured in four different configurations, as introduced in [15]. **(S1)** is configured for basic SyReC synthesis, **(S2)** is configured for line-aware synthesis, **(S3)** is configured for cost-aware synthesis, and **(S4)** is configured for both metrics best trade-off.

On the other hand, the VHDL-code has been synthesized using the two configurations: **(V1)** as described in Sections IV/V and with the improved realization **(V2)** as described in Section VI. **(V1)** result in the lowest cost among all scenarios, while **(V2)** results in a circuit with a minimal number of lines.

[3]This code is taken from (http://www.rfwireless-world.com).

[4]A SyReC benchmark (**lu_238.src**) in *RevLib* [24].

```
module lu(in op(2), out x0, inout x1, inout x2)
  if (op = 0) then
    x0 ^= (x1 & x2)
  else
    if (op = 1) then
      x0 ^= (x1 | x2)
    else
      if (op = 2) then
        x0 ^= (x1 ^ x2)
      else
        x0 ^= x1
        ~= x0
      fi (op = 2)
    fi (op = 1)
  fi (op = 0)
```

(a) SyReC description taken from $lu\_238.src$

```
entity alu is
  port( op   :  in unsigned (1 downto 0);
        x1,x2:  in bit_vector (31 downto 0);
        x0   :  out bit_vector (31 downto 0));
end entity test;

architecture data_flow of lu is
  begin
  x0<=  (x1 and x2) when (op = 0) else
        (x1 or  x2) when (op = 1) else
        (x1 xor x2) when (op = 2) else
        (not x1);
        end architecture data_flow;
```

(b) VHDL description

Fig. 10. HDL description of a basic 32-bit logic unit

This case shows that VHDL might compete or even overtake SyReC, when it comes to irreversible design problems.

TABLE II
EXPERIMENTAL RESULTS OF REALIZING 32-BIT LOGIC UNIT

Parameter	SyReC				VHDL	
	S1	S2	S3	S4	V1	V2
Gates	**384**	612	392	622	414	671
Total lines	197	**133**	198	134	235	**133**
Quantum cost	6557	10462	**2312**	3894	**682**	1207
Transistor cost	9856	15616	**6360**	10288	**3472**	5752

These two case studies show that the efficiency of each approach is problem-dependent. Hence, no clear winner can be declared within the scope of this study. Yet, we may still claim that this approach can offer, in some cases, an advantageous alternative to realize reversible circuits. In either case, VHDL might at least be considered as an acceptable alternative, for being more convenient for designers with no or little knowledge of the reversible computation paradigm.

VIII. CONCLUSIONS

In this work, we considered the conventional hardware description language VHDL for the synthesis of reversible circuits. A basic realization of VHDL code as well as possible ideas to improve the circuit measures have been discussed. The proposed approach has been discussed in comparison to a dedicated reversible HDL approach using the SyReC language. With these contributions, we provide an initial basis towards a VHDL-based reversible circuit design that requires only little knowledge of the reversible computation paradigm. The discussion shows that, despite having no clear winner between the two approaches, VHDL still provide the designers with a convenient alternative, and in some cases, more efficient design tool. In the future, we consider combining VHDL and SyReC in one integrated reversible circuit description environment.

ACKNOWLEDGEMENTS

This work has partially been supported by the EU COST Action IC1405.

REFERENCES

[1] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
[2] 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.
[3] A. De Vos, *Reversible Computing: Fundamentals, Quantum Computing and Applications*. Weinheim: Wiley-VCH, 2010.
[4] A. Rauchenecker, T. Ostermann, and R. Wille, "Exploiting reversible logic design for implementing adiabatic circuits," in *Int'l Conference on Mixed Design of Integrated Circuits and Systems*, 2017, pp. 264–270.
[5] R. Wille, R. Drechsler, C. Osewold, and A. G. Ortiz, "Automatic design of low-power encoders using reversible circuit synthesis," in *Design, Automation and Test in Europe*, 2012, pp. 1036–1041.
[6] R. Wille, O. Keszocze, S. Hillmich, M. Walter, and A. G. Ortiz, "Synthesis of approximate coders for on-chip interconnects using reversible logic," in *Design, Automation and Test in Europe*, 2016.
[7] A. Zulehner and R. Wille, "Taking one-to-one mappings for granted: Advanced logic design of encoder circuits," in *Design, Automation and Test in Europe*, 2017.
[8] L. G. Amarù, P. Gaillardon, R. Wille, and G. D. Micheli, "Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking," in *Design, Automation and Test in Europe*, 2016, pp. 175–180.
[9] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conf.*
[10] D. Große, R. Wille, G. W. Dueck, and R. Drechsler, "Exact multiple control Toffoli network synthesis with SAT techniques," *IEEE Trans. on CAD*, vol. 28, no. 5, pp. 703–715, 2009.
[11] K. Fazel, M. Thornton, and J. Rice, "ESOP-based Toffoli gate cascade generation," in *Pacific Rim Conference on Communications, Computers and Signal Processing*, 2007, pp. 206–209.
[12] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Design Automation Conf.*, 2009, pp. 270–275.
[13] C.-C. Lin and N. K. Jha, "RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits," *J. Emerg. Technol. Comput. Syst.*, vol. 10, no. 2, p. 14, 2014.
[14] R. Wille and R. Drechsler, "Effect of BDD optimization on synthesis of reversible and quantum logic," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 6, pp. 57–70, 2010.
[15] R. Wille, E. Schönborn, M. Soeken, and R. Drechsler, "SyReC: A hardware description language for the specification and synthesis of reversible circuits," *INTEGRATION, the VLSI Jour.*, vol. 53, pp. 39–53, 2016.
[16] M. K. Thomsen, "A functional language for describing reversible logic," in *Forum on Specification and Design Languages*, 2012, pp. 135–142.
[17] P. J. Ashenden, *The Designers Guide to VHDL*, 2008.
[18] A. Zulehner and R. Wille, "Make it reversible: Efficient embedding of non-reversible functions," in *Design, Automation and Test in Europe*, 2017.
[19] D. Maslov and G. W. Dueck, "Reversible cascades with minimal garbage," *IEEE Trans. on CAD*, vol. 23, no. 11, pp. 1497–1509, 2004.
[20] R. Wille, M. Soeken, D. M. Miller, and R. Drechsler, "Trading off circuit lines and gate costs in the synthesis of reversible logic," *INTEGRATION, the VLSI Jour.*, vol. 47, no. 2, pp. 284–294, 2014.
[21] R. Wille, M. Soeken, E. Schönborn, and R. Drechsler, "Circuit line minimization in the HDL-based synthesis of reversible logic," in *IEEE Annual Symposium on VLSI*, 2012, pp. 213–218.
[22] A. Barenco, C. H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *The American Physical Society*, vol. 52, pp. 3457–3467, 1995.
[23] B. Desoete and A. D. Vos, "A reversible carry-look-ahead adder using control gates," *INTEGRATION, the VLSI Jour.*, vol. 33, no. 1-2, pp. 89–104, 2002.
[24] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: an online resource for reversible functions and reversible circuits," 2008, pp. 220–225, RevLib is available at http://www.revlib.org.