Designing a RISC CPU in Reversible Logic

Robert Wille Mathias Soeken Daniel Große Eleonora Schönborn Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{rwille,msoeken,grosse,eleonora,drechsle}@informatik.uni-bremen.de

***Abstract*—Driven by its promising applications, reversible logic received significant attention. As a result, an impressive progress has been made in the development of synthesis approaches, implementation of sequential elements, and hardware description languages. In this paper, these recent achievements are employed in order to design a RISC CPU in reversible logic that can execute software programs written in an assembler language. The respective combinational and sequential components are designed using state-of-the-art design techniques.**

I. Introduction

With increasing miniaturization of integrated circuits, the reduction of power dissipation has become a crucial issue in today's hardware design process. While due to high integration density and new fabrication processes, energy loss has significantly been reduced over the last decades, physical limits caused by Landauer's principle will remain [1]. Landauer observed that independently from the applied technology $\ln 2 \cdot kT$ Joule of energy is dissipated for each lost bit of information (k is the Boltzmann constant and T is the temperature). That is, each time e.g. an AND operation is performed, where two input values are transformed into a single one, this fraction of energy is dissipated. While this amount has been negligible in the past, in particular for future designs it becomes an increasingly important factor [2].

In order to avoid this physical limitation, researchers propose reversible circuits as a promising alternative. Already in the 1970s, Bennett showed that zero energy dissipation is only possible, if information-lossless computation is performed [3]. This applies to reversible circuits, since they map each input pattern uniquely to a certain output pattern. First physical realizations confirming this observation (e.g. in terms of a reversible CMOS 4-bit adder) have already been presented [4].

Driven by these results as well as by further promising applications (e.g. in the domain of quantum computation [5]), researchers started to apply these concepts on larger circuitries. *Central Processing Units* (CPUs) – as the core element of many systems – received thereby particular attention (see e.g. [6], [7], or recently [8]). However, either hand-crafted architectures or processing units with a very small set of supported operations have been presented so far.

A reason for this has been the lack of efficient tools and methods for the design of complex reversible circuits. But in the last years, an impressive progress has been made in the development of approaches for synthesis [9], [10], [11], [12], the implementation of reversible sequential elements [13], [14], [15], [16], or, recently, the introduction of hardware description languages [17].

In this paper, the recent progress in the field of reversible circuit design is employed in order to design a complex system, i.e. a RISC CPU composed of reversible gates. Starting from a textual specification, first the core components of the CPU are identified. Previously introduced approaches are applied next to realize the respective combinational and sequential elements. More precisely, the combinational components are designed using the reversible hardware description language SyReC [17], whereas for the realization of the sequential elements an external controller (as suggested in [16]) is utilized.

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

Therewith, the contribution of the paper is twofold. On the one hand, a comprehensive case study is provided, showing the application of state-of-the-art design techniques in order to design a large system. Considering that some years ago, automatic synthesis approaches were applicable to functions given in terms of truth-tables only (see e.g. [9]), this is a major step towards the design of high-end systems in reversible logic. On the other hand, with the resulting reversible CPU, a non-trivial reversible circuit becomes available, which can serve as benchmark for other areas such as verification or test[1].

The paper is structured as follows. The next section introduces the basics of reversible circuits as well as the hardware description language SyReC, which is used to implement the proposed CPU. Afterwards, the specification of the CPU is provided in Section III, while Section IV discusses the implementation details. Section V demonstrates the execution of a software program on the proposed CPU. Finally, conclusions are given in Section VI.

II. Background

This section introduces the basics of reversible circuits. Furthermore, the SyReC hardware description language is reviewed, which is used later to specify core components of the CPU. The descriptions are kept brief. For a more detailed treatment, we refer to the respective references.

A. Reversible Circuits

Reversible circuits realize functions with a unique input/output mapping, i.e. bijections. A reversible circuit is composed as a cascade of reversible gates [5]. For a set of variables

[1] To this end, the CPU has been made public available at RevLib [18].

0195-623X/11 $26.00 © 2011 IEEE
DOI 10.1109/ISMVL.2011.39

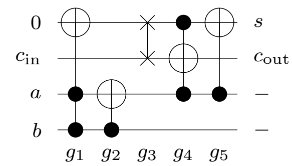


Fig. 1. Reversible circuit realizing a full adder

$X = \{x_1, \ldots, x_n\}$, a *reversible gate* has the form $g(C,T)$, where $C = \{x_{i_1}, \ldots, x_{i_k}\} \subset X$ is the set of *control lines* and $T = \{x_{j_1}, \ldots, x_{j_l}\} \subset X$ 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. In this work, circuits realized by Toffoli gates [19] and Fredkin gates [20] are considered. A Toffoli gate has a single target line x_j and maps the input $(x_1, x_2, \ldots, x_j, \ldots, x_n)$ to the output $(x_1, x_2, \ldots, x_{i_1}x_{i_2} \cdots x_{i_k} \oplus x_j, \ldots, x_n)$. That is, a Toffoli gate inverts the target line if and only if all control lines are assigned to 1. A Fredkin gate has two target lines x_{j_1} and x_{j_2}. The gate interchanges the values of the target lines if and only if the conjunction of all control lines evaluates to 1.

By definition, reversible circuits can only realize reversible functions. Thus, in order to realize non-reversible functions, a process called *embedding* has to be performed prior to the synthesis. Therefore, additional garbage (i.e. don't care) outputs and constant inputs are added to embed the non-reversible function into a reversible one [21]. Besides that, constant inputs and garbage outputs are also used frequently in order to realize larger functions (see e.g. [12], [17]).

As an example, Fig. 1 shows a reversible circuit realization of an 1-bit adder. Since the adder is a non-reversible function, one additional constant input and two garbage outputs are used to realize this function as a reversible circuit. The gates g_1, g_2, g_4, and g_5 are thereby Toffoli gates, while gate g_3 is a Fredkin gate.

B. The SyReC HDL

SyReC is a hardware description language for reversible circuits proposed in [17] and is based on the reversible software language Janus [22]. It provides fundamental constructs to define control and data operations, while still preserving reversibility.

In Fig. 2, the syntax of SyReC is outlined. Each SyReC program (denoted by P) consists of signal declarations (denoted by D) and procedure declarations (representing sub-circuits). Signals can hold non-negative integer values and are identified by strings. The bit-width of a signal can optionally be defined (if not, a default bit-width is applied). Constants are denoted by c. Each procedure consists of a name (id) and a sequence of statements (denoted by S) including operations, reversible conditionals, reversible loops, as well as call and uncall of procedures (Lines 5 to 9 in Fig. 2). The number of iterations in loops has to be available prior to the compilation, i.e. dynamic

(1) P ::= D* (**procedure** *id* S⁺)⁺
(2) D ::= *x* | *x* (*c*)
(3) V ::= *x* | *x.N*:*N* | *x.N*
(4) N ::= *c* | **#**V
(5) S ::= V **<=>** V | V ⊕**=** E |
(6) **if** E **then** S **else** S **fi** E |
(7) **from** *N* **do** S **loop** S **until** *N* |
(8) **for** *N* **do** S **until** *N* |
(9) **call** *id* | **uncall** *id* | **skip**
(10) E ::= *N* | V | (E ⊙ E) | (E ⦸ *N*)
(11) ⊕ ::= **+** | **-** | **^**
(12) ⊙ ::= ⊕ | ***** | **/** | **%** | ***/** | **&** | **|** | **&&** | **||** |
(13) **<** | **>** | **=** | **!=** | **<=** | **>=**
(14) ⦸ ::= **<<** | **>>**

Fig. 2. Syntax of the hardware language SyReC

loops are not allowed. Signals within statements are denoted by V allowing access to the whole signal (*x*), a certain bit (*x.N*), or a range of bits (*x.N*:*N*). The bit-width of a signal can also be accessed (**#**V).

A distinction is made between *reversible assignment operations* (denoted by ⊕) and not necessarily reversible *binary operations* (denoted by ⊙). The former ones 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 (^=), since they preserve the reversibility (i.e. it is possible to compute these operations in both directions).

In contrast, binary operations, i.e. arithmetic (+, *, /, %, */), bit-wise (&, |, ^), logical (&&, ||), relational (<, >, =, !=, <=, >=), and shifting (<< | >>) operations, may not be reversible. Thus, they can only be used in right-hand expressions which preserve (i.e. do not modify) 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 *a***b* in SyReC, a new free signal *c* must be introduced which is used to store the product. That results in the expression *c*^=*a***b*. In comparison to common (irreversible) programming languages, statements such as *a*=*a***b* are not allowed. Using SyReC, complex reversible circuits can be specified. An example of a circuit specified in SyReC is given in Section IV-B.

III. Specification of the CPU

In this section, the basic data of the proposed RISC CPU is provided. The specification is inspired by the design of a conventional CPU (see [23]). The CPU was created in order to execute software programs provided in terms of the assembler language shown in Table I. This includes

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

The respective assembler programs are transformed into sequences of binary instruction words, which are processed by the CPU. A single instruction word is specified as shown

TABLE I
ASSEMBLER INSTRUCTIONS

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

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

Instruction format:

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

Fig. 3. Instruction word representing an ADD instructions

in Fig. 3 by means of the ADD operation. Since in total 25 different instructions are supported, the opcode consists of the five most significant bits of the instruction word (00111 in case of the ADD instruction). The remaining bits give the encoding of the natural numbers i, j, and k, which address the respective registers used by the instruction.

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

IV. IMPLEMENTATION OF THE CPU

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

A. Overview

Fig. 4 provides a schematic overview showing the implementation of the proposed CPU. In the following, the respective components are briefly described from the left-hand side to the right-hand side.

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

1) If an arithmetic or logical operation is performed, the respective operands are extracted from the instruction word and assigned to the signals op1 and op2, respectively. These two signals together with oprt, which defines the respective operation, are passed to the *ALU*. Besides that, the signal write is assigned a logic value 1 indicating that the result of the operation should be stored in a target register addressed by dest. Finally, the signal inc is set to 1, indicating that the program counter has to be increased by 1.
2) If instead a control operation (e.g. a JMP) is performed, the signals op1, op2, oprt, write, and dest are not required for further operation in the current cycle, whereas the signal inc is assigned a logic value 0. Further, jmp is set to the new address of the program memory depending on the instruction word.
3) A memory access using load and store instructions can be conducted directly by the control unit. In case of an LDD instruction, the data is fetched from the memory and stored in the respective register by adjusting the corresponding signal register. In contrast, in case of an STO instruction, the value of the source register is read and stored in the respective memory address. All other signals are assigned, such that the results of the components are not used (in case of the ALU) or remain unchanged (in case of register file). Also here, signal inc is assigned to logic 1.

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

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

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

- Combinational components, i.e. the circuit elements needed to perform the actual computation. This includes the control unit, the ALU, the program counter, and the register file, respectively. That is, all shaded components in Fig. 4 fall in this category.

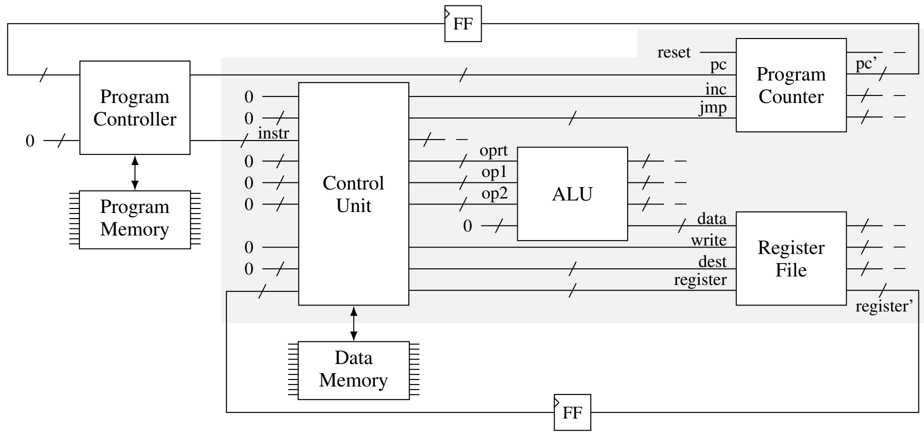


Fig. 4. Schematic diagram of the CPU implementation

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

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

B. Combinational Components

In order to realize combinational reversible circuits, a wide range of synthesis approaches have been introduced in the recent years (see e.g. [9], [10], [11], [12]). Most of them rely on Boolean descriptions such as truth tables or *Binary Decision Diagrams* (BDDs). But since the CPU includes complex operations (e.g. large control paths and arithmetic operations), we used the SyReC programming language as well as its respective synthesis engine to realize the combinational components of the CPU [17].

Thus, the control unit, the ALU, and the program counter can be implemented on a higher level of abstraction. This avoids scalability problems, which would occur if truth-table-based or BDD-based approaches were applied. In contrast, hierarchical synthesis approaches (such as the SyReC engine) tend to generate circuits with a large number of constant inputs. This can partially be improved by post-synthesis optimization approaches (e.g. [24]), but still remains an open problem, which is left for future work. Besides that, new design paradigms have to be considered.

As an example, the SyReC code of the program counter is given in Fig. 5(a). One new design paradigm becomes already evident in this example. According to the specification, the program counter should be assigned 0, if the primary input `reset` is assigned 1. Due to a lack of conventional assignment operations which would destroy the reversibility, this is realized by a new additional signal (denoted by `zero` and set to 0) as well as a SWAP operation (see Line 6 of Fig. 5(a)). Similar design decisions have to be made e.g. to realize the desired control path or to implement the respective functionality of the ALU. In contrast, the increase of the program counter is a reversible operation and, thus, can easily be implemented by the respective += instruction (Line 9).

The resulting circuit generated by the SyReC synthesizer is shown in Fig. 5(b). Note that the bit-widths of the signals are scaled down to 2 in order to improve the readability. The first two lines give the current value of the program counter (`pc_1`, `pc_0`), while the same lines on the right-hand side hold the next state values (`pc_1`′, `pc_0`′) used as inputs for the flip-flops as depicted in Fig. 4.

The remaining combinational components are realized similarly. However, due to page limitation and size restrictions, the complete SyReC code as well as the resulting circuits of all combinational components cannot be provided in this paper. The sources are completely available on RevLib [18].

C. Sequential Components

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

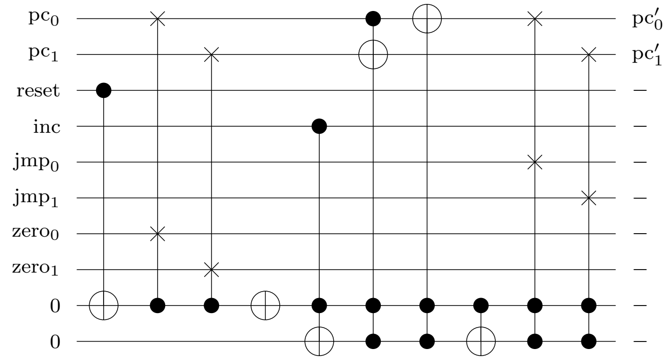
The first paradigm (suggested e.g. in [16]) arguments that a reversible circuit retains in its state as long as its signal values remain unchanged. Thus, a combinational circuit can be treated as a core component of a sequential device. More precisely, using e.g. a classical (non-reversible) controller, output values from one cycle are applied to the respective input signals of the next cycle. Therefore, the clocking as well as the feedback is handled by the controller, while the actual computation is performed on a combinational reversible circuit.

The second paradigm considers the realization of the sequential elements directly in reversible logic. For this purpose, several suggestions on how to realize the respective memory elements as flip-flops, latches, or registers have been made (see e.g. [13], [14], [15]). Using these basic sequential elements, more complex sequential components can easily be constructed.

```
pc ( 2 ) reset ( 1 )
inc ( 1 ) jmp ( 2 ) zero ( 2 )

procedure pc
if ( reset = 1 ) then
    pc <=> zero
else
    if ( inc = 1 ) then
        pc + = 1
    else
        pc <=> jmp
    fi ( inc = 1 )
fi ( reset = 1 )
```

(a) SyReC code



(b) Resulting circuit

Fig. 5. Implementation of the program counter (scaled down to a bit-width of 2)

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

D. Characteristics of the Resulting Circuit

Using the schematic diagram described in Fig. 4 and by plugging the synthesized combinational parts together, a reversible circuit results, composed of 1,139 circuit lines (including 867 lines with constant inputs), 5,047 Toffoli gates, and 1,692 Fredkin gates. Considering established cost metrics, this circuit has transistor costs of 504,904 (see [25] for more details on transistor costs) and quantum costs of 501,119 (see [26] for more details on quantum costs)[2]. Together with the external controller for the sequential components, this reversible circuit represents a CPU ready for running programs.

V. Executing Programs on the CPU

With the CPU implemented as described in the previous sections, arbitrary software programs composed of the assembler instructions given in Table I can be executed. Therefore, first an assembler program is translated into a sequence of respective instruction words by applying techniques proposed in [23]. Afterwards, the resulting instruction words are loaded into the program memory, while the data memory is initialized with desired values. Both, the program memory and the data memory, are realized by an external controller implemented in terms of a Python script. Overall, this allows to run translated object code, i.e. a sequence of instruction words.

The execution of a program on the proposed CPU is illustrated using the assembler program depicted in Fig. 6. Here, the sequence of Fibonacci numbers defined by $f(n) = f(n-1) + f(n-2)$ with $f(0) = f(1) = 1$ is computed. More precisely, the program generates the Fibonacci number $f(n+1)$, whereby $n > 1$ is given in the register *R*[7]. The result is stored in *R*[4].

The waveform obtained by simulating this program (with $n = 4$) on the CPU is given in Fig. 7. The identifiers *clk*,

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

Fig. 6. Assembler program for Fibonacci number computation

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. The rows *R*[2], *R*[3], *R*[4], and *R*[7] list the values of the respective registers. For the sake of clarity, all other signal values are omitted. Note that the value of the program counter always corresponds to the respective line number of the code given in Fig. 6. In each time frame always the updated values of the signals obtained after the execution are listed.

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

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

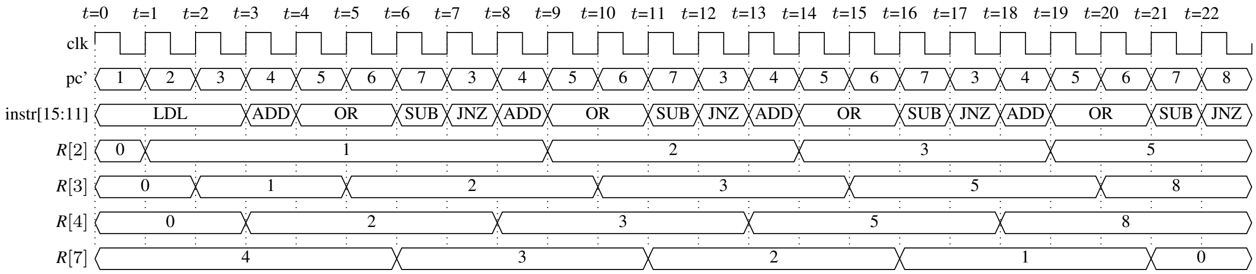


Fig. 7. Waveform illustrating the execution of the program given in Fig. 6

VI. Conclusion

In this paper, we proposed a design of a RISC CPU realized using reversible logic. Therefore, recent achievements in the domain of reversible circuit design have been employed. In particular, this includes the hardware description language SyReC, which has been used to design the combinational components of the CPU. In contrast, the sequential components have been realized using an external controller. With the proposed CPU, it is possible to execute software programs using an assembler language. Besides that, the circuit can be used as benchmark for other areas such as the verification or the test of reversible circuits. Therefore, the CPU has been made public available at RevLib [18].

Future work is focused on the optimization of the resulting circuit. As discussed in Section IV-B, in particular reducing the number of lines is important. For this purpose, one could consider the approach presented in [24]. Furthermore, having the CPU design available, a physical realization of a complex application is possible. So far, only simple circuits have been physically realized. Finally, the design of a CPU processing reversible software languages (as e.g. Janus [22]) may provide an interesting case study.

Acknowledgments

The authors would like to thank Bastian Blachetta for his help during the implementation of the CPU. This work was supported by the German Research Foundation (DFG) (DR 287/20-1).

References

[1] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM J. Res. Dev.*, vol. 5, p. 183, 1961.

[2] 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.

[3] C. H. Bennett, "Logical reversibility of computation," *IBM J. Res. Dev*, vol. 17, no. 6, pp. 525–532, 1973.

[4] 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.

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

[6] J. S. Hall, "A reversible instruction set architecture and algorithms," in *Workshop on Physics and Computation*, 1994, pp. 128–134.

[7] C. J. Vieri, "Pendulum: A reversible computer architecture," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[8] M. K. Thomsen, R. Glück, and H. B. Axelsen, "Reversible arithmetic logic unit for quantum arithmetic," *Journal of Physics A: Mathematical and Theoretical*, vol. 43, no. 38, 2010.

[9] 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.

[10] P. Kerntopf, "A new heuristic algorithm for reversible logic synthesis," in *Design Automation Conf.*, 2004, pp. 834–837.

[11] 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.

[12] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Design Automation Conf.*, 2009, pp. 270–275.

[13] H. Thapliyal and M. B. Srinivas, "A beginning in the reversible logic synthesis of sequential circuits," in *MAPLD Int'l Conf.*, 2005.

[14] M. Chuang and C. Wang, "Synthesis of reversible sequential elements," in *ASP Design Automation Conf.*, 2007, pp. 420 –425.

[15] N. M. Nayeem, M. A. Hossain, L. Jamal, , and H. Babu, "Efficient design of shift registers using reversible logic," in *Int'l Conf. on Signal Processing Systems*, 2009, pp. 474–478.

[16] M. Lukac and M. Perkowski, "Quantum finite state machines as sequential quantum circuits," in *Int'l Symp. on Multi-Valued Logic*, 2009, pp. 92–97.

[17] 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.

[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] T. Toffoli, "Reversible computing," in *Automata, Languages and Programming*, W. de Bakker and J. van Leeuwen, Eds. Springer, 1980, p. 632, technical Memo MIT/LCS/TM-151, MIT Lab. for Comput. Sci.

[20] E. F. Fredkin and T. Toffoli, "Conservative logic," *International Journal of Theoretical Physics*, vol. 21, no. 3/4, pp. 219–253, 1982.

[21] 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.

[22] 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.

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

[24] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Design Automation Conf.*, 2010.

[25] M. K. Thomson and R. Glück, "Optimized reversible binary-coded decimal adders," *J. of Systems Architecture*, vol. 54, pp. 697–706, 2008.

[26] 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.