A Reversible Processor Architecture and Its Reversible Logic Design

Michael Kirkedal Thomsen, Holger Bock Axelsen, and Robert Glück

DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen, Denmark
{shapper,funkstar}@diku.dk, glueck@acm.org

Abstract. We describe the design of a purely reversible computing architecture, Bob, and its instruction set, BobISA. The special features of the design include a simple, yet expressive, locally-invertible instruction set, and fully reversible control logic and address calculation. We have designed an architecture with an ISA that is expressive enough to serve as the target for a compiler from a high-level structured reversible programming language.

All-in-all, this paper demonstrates that the design of a complete reversible computing architecture is possible and can serve as the core of a programmable reversible computing system.

1 Introduction

Energy consumption is an important aspect of most computing systems today and this is especially true for embedded systems and battery-dependent computers. Reversible computing has the potential to reduce power consumption and heat dissipation [11, 14].

The design of reversible computing systems and programs is, however, not a trivial extension of the conventional case. Not all problems have simple reversible implementations and rethinking the entire problem might be needed for a solution. Reversible programming languages [15, 28] have special constructs (*e.g.* an *if-then-else* statement also needs a joining *assertion* that verifies the computational path) which complicates program development, and the need for new programming methodologies is evident.

There are, however, specific domains that are clear-cut for reversible computing: lossless discrete transformations like FFT and wavelets [10] used in compression and analysis of multimedia signals, or simulation of physical systems. Developments in areas from low-level circuit design [9] and synthesis [16, 19, 25] to high-level languages such as Janus [27,28] and compilers [1] have also provided more insight to the design of reversible systems.

Fully reversible computing systems[1] are still years of development away from general purpose computers. In this paper we show the design of a simple re-

[1] A first outline of this fully reversible (both abstract machine and implementation) architecture was presented in Thomsen, Glück, Axelsen, *Towards Designing a Reversible Processor Architecture*, *work-in-progress*, at the *1st Workshop on Reversible Computation, 2009* in York.

A. De Vos and R. Wille (Eds.): RC 2011, LNCS 7165, pp. 30–42, 2012.

versible computing architecture for a reversible implementation of a Harvard architecture, Sect. 2. The architecture has a small instruction set but is still powerful enough to be Turing-complete in a reversible sense [3] and expressive enough to be the target for a compiler [1] from the high-level language Janus, Sect. 3.

The low-level implementation of Bob is designed with elementary reversible logic gates [5, 13] resulting in a robust technology-independent design, Sect. 4. It makes use of an extended version of the latest design of reversible arithmetic logic units [21], Sect. 4.1, and has a novel control structure that simplifies the address calculation compared to previous approaches [12,24], Sect. 4.2. As memory in reversible hardware is still an open question, we shall assume memory that is operationally reversible, such that the design is independent of any actual future memory implementation, regardless of whether this is based on conventional volatile memory [6] or reversible models like the *rotary element* [17]. For verification we implemented the design in Verilog. The programming was self-restricted to uphold the conventions of reversible logic design, Sect. 4.3.

2 The Problem of Control

In this section we describe the control logic used in our Harvard architecture, and the reasoning behind it.

In a conventional processor architecture, the address of the next instruction to be executed is often found by overwriting the program counter with a static address. As a result, the information about the old program counter is erased. If this is the case, then we do not know how to make a backwards step, *i.e.* irreversibility.

A solution to this problem could be to use the Landauer embedding [14] and generate a *trace* of all previous program counters. This approach, suggested by Cezzar [7], is not satisfactory. The trace, which would be as long as the number of executed instructions, is not part of the program's desired result and is an extremely wasteful use of memory. A processor which accumulates more and more garbage in this fashion is not practical.

Instead of using only a single register for program control (the program counter), we shall use an approach developed for the reversible von Neumann architecture Pendulum [12,24] as formalized in [4], where the address calculation of the reversible abstract machine relies on *three* special-purpose registers:

- *program counter* (*pc*): points at the current instruction in memory,
- *branch register* (*br*): contains information about the offset from the current to the next instruction, and
- *direction bit* (*dir*): specifies the current direction of execution; either FALSE (forward) or TRUE (backward).

The calculation of the next program counter (*pc*) now only depends on the branch register (*br*) and the direction bit (*dir*). If the value of the branch register is *zero*, then the execution will continue to the next instruction by adding 1

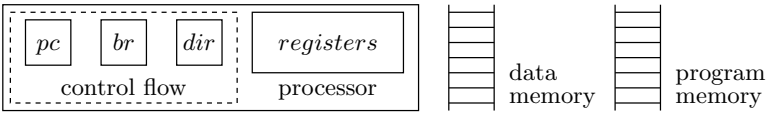

Fig. 1. The reversible Harvard architecture

to (or subtracting 1 from) the program counter, depending on the execution direction given by the direction bit. If the branch register contains a *non-zero* value then this is added to (or subtracted from) the program counter. In both cases the program counter is reversibly updated, and the branch register and direction bit are preserved. We therefore have enough information to do the *inverse* calculation to determine the previous instruction, *i.e.* reversibility.

Figure 1 shows the abstract reversible Harvard architecture. While a von Neumann architecture has only one memory containing both the program and data, they are separated in a Harvard architecture. This separation simplifies the reversible model by ensuring that a memory instruction cannot update its own instruction cell, which would lead to irreversibility.

3 A Simple Instruction Set Architecture, BobISA

The choice of the instruction set influences not only the expressiveness of the assembly language, but also the costs of the underlying hardware realization. A larger instruction set with many complex operations can increase the expressiveness and reduce code size, but will also result in higher costs in terms of gates, logic depth, ancillae, and so forth. We require that the reversible instruction set is *r-Turing complete* [3]; meaning that it can implement an interpreter for reversible Turing Machines without the use of a history, or other garbage [2]. Furthermore, all instructions are required to be *reversible updates* [4] and locally invertible. The rest of this section describes the 17 instructions of BobISA, divided into three types: arithmetic/logic instructions, branch instructions, and memory instructions.

3.1 Arithmetic-Logic Instructions

Table 1 shows the set of reversible arithmetic/logic instructions. It includes addition (`ADD` and `ADD1`), subtraction (`SUB` and `SUB1`), negation (`NEG`), and exclusive-or (`XOR` and `XORI`); the immediate instruction, `XORI`, computes exclusive-or with a given constant value. These are the basic instructions included in our reversible ALU design [21]. To ensure reversibility we use modular arithmetic.

Conventional processors typically allow multiplication and division by 2, implemented as left or right shifts. These are irreversible operations (*e.g.* the division by 2 deletes the least significant bit), so to circumvent this, left and right *roll* operations could be used instead. Here, we propose a novel solution with

Table 1. Arithmetic-logic instructions, their inverses and effect on registers R

i	$Inv(i)$	$Effect(i)$
`ADD` reg_d reg_s	`SUB`	$R(reg_d) \leftarrow R(reg_d) +_n R(reg_s)$
`SUB` reg_d reg_s	`ADD`	$R(reg_d) \leftarrow R(reg_d) -_n R(reg_s)$
`ADD1` reg_d	`SUB1`	$R(reg_d) \leftarrow R(reg_d) +_n 1$
`SUB1` reg_d	`ADD1`	$R(reg_d) \leftarrow R(reg_d) -_n 1$
`NEG` reg_d	`NEG`	$R(reg_d) \leftarrow 0 -_n R(reg_d)$
`XOR` reg_d reg_s	`XOR`	$R(reg_d) \leftarrow R(reg_d) \oplus R(reg_s)$
`XORI` reg_d imm	`XORI`	$R(reg_d) \leftarrow R(reg_d) \oplus imm$
`MUL2` reg_d	`DIV2`	$R(reg_d) \leftarrow mul2_n(R(reg_d))$
`DIV2` reg_d	`MUL2`	$R(reg_d) \leftarrow div2_n(R(reg_d))$

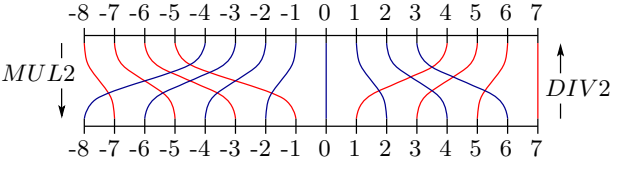

Fig. 2. Example of division (`DIV2`) and multiplication (`MUL2`) by 2 with 4-bit two's complement numbers. The solid blue lines show the inputs that are well defined.

a division/multiplication by 2 that conserves the sign of the two's complement numbers, but only returns the division or multiplication by 2 if the input is well-defined. For division, only the even numbers return the input value divided by 2, and multiplication only returns the input value multiplied by 2 if the input is small enough for it to double without overflow. The rest of the input values we map to values such that reversibility and local invertibility is assured and the instructions are easy to implement in logic. For a more intuitive description, Fig. 2 shows the mapping for 4-bit two's-complement numbers.

The multiplication/division operations are formally defined as

$$mul2_n(x) = \begin{cases} x \cdot 2 & \text{if } -2^{n-2} \leq x < 2^{n-2}, \\ x \cdot 2 - 2^n + 1 & \text{if } x \geq 2^{n-2}, \\ x \cdot 2 + 2^n + 1 & \text{if } x < -2^{n-2}, \end{cases} \tag{1}$$

and

$$div2_n(x) = \begin{cases} x/2 & \text{if } x \text{ is even}, \\ \frac{x-1}{2} + 2^{n-1} & \text{if } x \text{ is odd, and } x > 0, \\ \frac{x-1}{2} - 2^{n-1} & \text{if } x \text{ is odd, and } x < 0, \end{cases} \tag{2}$$

where n is the number of bits used in the representation of x.

A general restriction in reversible programming languages is that a register (or variable) must only be updated with a source value does not come from the register itself. (*E.g.* $a \leftarrow a - a$ is not allowed.) A violation of this will result in information destruction. The standard way of resolving this is by checking that the destination register reg_d and (second) source register reg_s are syntactically different. This check is simple at high abstraction levels, but implementing it at the logic level results in a large overhead. Also, if this check *fails* the whole program execution fails; it is hard at the logic circuit level to define the meaning of a failing architecture execution.

Instead, our solution is to slightly alter the memory model of the registers. When register reg_d is read, its value is swapped with 0. Now, if reg_s is the same as reg_d then the value of reg_s becomes 0 instead of the original value of reg_d, and so the value of register reg_d will not be destroyed. (*E.g.* $a := a - 0$ is calculated instead of $a := a - a$.) Writing values back to the register file is done in the opposite order: first the value of reg_s is swapped into the register reg_s, then afterwards the same for reg_d. In both writing cases the auxiliary value that is swapped in/out of the registers is 0.

3.2 Branch Instructions

Branch instructions are needed for control flow in programs. For BobISA, we have chosen four conditional branch, one unconditional branch, and two special swap-branch-register instructions for the instruction set (see Table 2).

The four conditional branch instructions were chosen for their simple implementation. Because we use two's complement numbers, both *greater-than-or-equal-to-zero* (`BGEZ`) and *less-than-zero* (`BLZ`) are simple checks of the most significant bit of the value in reg_d (FALSE for values greater than or equal to zero and TRUE if the value is less than zero). The other two conditional instructions are branches on *even* (`BEVN`) and *odd* (`BODD`) numbers. These are determined by a simple check of the *least* significant bit: a FALSE implies an even number and a TRUE implies an odd number. For all four instructions, if the branch condition evaluates to true then the offset (*off*) is *added* to the branch register; else, the branch register is left unchanged. There is also an unconditional branch instruction (`BRA`) that always updates the branch register with the given offset.

In conventional ISAs a *jump-and-link* instruction is used for procedure calls; it stores the current program counter in a register as a *return address* and then jumps to a given address. We know for Bob that the program counter is only updated by the branch register, so we can simulate the jump-and-link by loading an offset into the branch register, performing the jump and then have an instruction at the target address saving the branch register as a *return offset*.

There are two special instructions to support this: `SWB` and `RSWB`. The *swap-branch-register* (`SWB`) will swap the value of a given register with the value of the branch register. The *swap-branch-register-and-reverse* (`RSWB`) will do the same, and furthermore reverse the execution direction by flipping the direction bit.

This can be used for inverse procedure calls. The use of the RSWB instruction is novel and was chosen because it simplifies the logic for the *pc* update significantly, reducing the gate count and logic depth compared to previous designs [12, 24, 4].

3.3 Memory Instruction

The usual *load/store* memory instructions in conventional instruction sets are, by themselves, irreversible. However, by combining the load and the store instructions into a single *exchange* (EXCH) instruction, the result is a memory instruction that is reversible and self-inverse (as shown in Table 2), cf. [12, 24]. This takes a register (reg_d) that contains some value to be exchanged into memory and a register (reg_a) that contains the address of the cell in memory that we want to exchange, as arguments. The value in the register and the value at the address in the memory are then swapped.

Table 2. Branch and memory instructions, their inverses, and effect on the general purpose registers R, special purpose registers br and dir, and data memory M

i	$Inv(i)$	$Effect(i)$
BGEZ reg_d off	BGEZ reg_d $-off$	$br \leftarrow br +_n (R(reg_d) \geq 0 \texttt{ ? } off \texttt{ : } 0)$
BLZ reg_d off	BLZ reg_d $-off$	$br \leftarrow br +_n (R(reg_d) < 0 \texttt{ ? } off \texttt{ : } 0)$
BEVN reg_d off	BEVN reg_d $-off$	$br \leftarrow br +_n (even(R(reg_d)) \texttt{ ? } off \texttt{ : } 0)$
BODD reg_d off	BODD reg_d $-off$	$br \leftarrow br +_n (odd(R(reg_d)) \texttt{ ? } off \texttt{ : } 0)$
BRA off	BRA $-off$	$br \leftarrow br +_n off$
SWB reg_d	SWB reg_d	$br \leftrightarrow R(reg_d)$
RSWB reg_d	RSWB reg_d	$br \leftrightarrow R(reg_d)$; $dir \leftarrow \neg dir$
EXCH reg_d reg_a	EXCH reg_d reg_a	$R(reg_d) \leftrightarrow M(R(reg_a))$

4 The Architecture of the Reversible Machine, Bob

Based on the ISA above we design an architecture, called *Bob*, that performs one instruction within a single clock-cycle. We have chosen a 16-bit architecture, which leads to the following design properties and the defined instruction encoding shown in Fig. 3.

- *Registers* - 4 bits for register numbering allows for 16 registers in total, each with a size of 16 bits. Using two's complement representation, numbers can range from -32768 through 32767.
- *Memory* - We can index 2^{16} words of 16 bits (the maximum size we can load into the registers). This gives a memory cap of 128 KB.
- *Jumps* - With an offset length of 8 bits, a branch-jump can not be of more than 127 lines. However, jumps can be arbitrarily long by the using the SWB instruction.
- *Immediates* - With 8 bits, immediate values must range from -128 to 127.

bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Arith & mem	*opcode*				reg_d				reg_s				*arith*			
Branch & imm	*opcode*				reg_d				*off/imm*							
`ADD`	1	1	0	0	reg_d				reg_s				0	1	0	0
`SUB`	1	1	0	0	reg_d				reg_s				1	1	0	1
`ADD1`	1	1	0	0	reg_d				0	0	0	0	0	1	1	0
`SUB1`	1	1	0	0	reg_d				0	0	0	0	1	1	1	1
`NEG`	1	1	0	0	reg_d				0	0	0	0	0	1	1	1
`XOR`	1	1	0	0	reg_d				reg_s				0	0	0	0
`XORI`	0	0	0	0	reg_d				*imm*							
`MUL2`	1	0	1	0	reg_d				0	0	0	0	0	0	0	0
`DIV2`	1	0	0	1	reg_d				0	0	0	0	0	0	0	0
`EXCH`	1	0	0	0	reg_d				reg_a				0	0	0	0
`BGEZ`	0	0	1	1	reg_d				*off*							
`BLZ`	0	0	1	0	reg_d				*off*							
`BEVN`	0	1	0	1	reg_d				*off*							
`BODD`	0	1	0	0	reg_d				*off*							
`BRA`	0	0	0	1	0	0	0	0	*off*							
`RSWB`	0	1	1	1	reg_d				0	0	0	0	0	0	0	0
`SWB`	0	1	1	0	reg_d				0	0	0	0	0	0	0	0

Fig. 3. Instruction formats and instruction set encoding for Bob

– *Register zero* - Register 0, reg_0 is assumes to always contain the value 0. Instructions with only one register (`NEG`, `ADD1`, etc.) are implemented with this requirement in mind (*e.g.* `ADD1` reg_d is implemented as $R(reg_d) \leftarrow R(reg_d) +_n R(reg_0) +_n 1$).[2]

While it is a primary requirement for us to keep the implementation *garbage-free*, we also try to reduce the number of ancillae bits, and keep circuit size at a minimum. We therefore accept that the delay of sub-circuits (*e.g.*, the ALU and adders) are linear with respect to the number of input bits, as this lowers the above costs.

Figure 4 shows a detailed design of the processor, and Table 3 shows the gate count. Even though it has many similarities with the MIPS R2000 processor [18], there are some significant differences. Notice, for example, that preserving information everywhere implies that the control signals from the control logic unit can not be deleted, but have to be uncomputed using an *inverse control logic unit*. Other significantly different parts are the *arithmetic logic unit* and the *address calculation logic*, which will be described below.

[2] Breaking the assumption about register 0 will *not* break reversibility of the architecture, but only result it a processor that do not behave as expected; *e.g.* the example with `ADD1`.

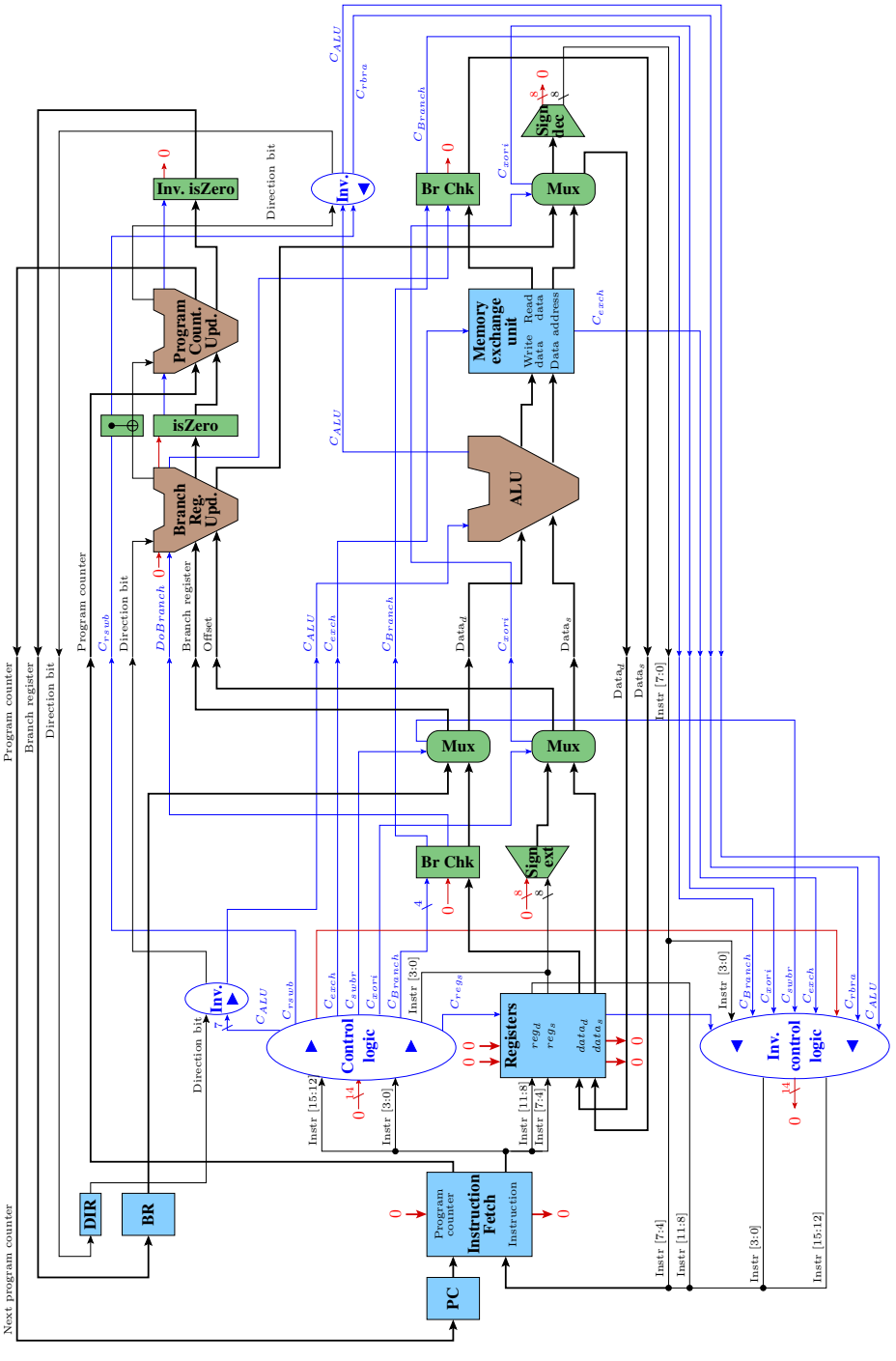

Fig. 4. The logic design of the reversible processor. The black dots indicate split and merge of lines, *not* fan-out and fan-in. The small blue arrows indicates input and output control lines. The light blue boxes are memory elements, the brown polygons are the ALU and other adders, and the small green figures are minor combinational circuits.

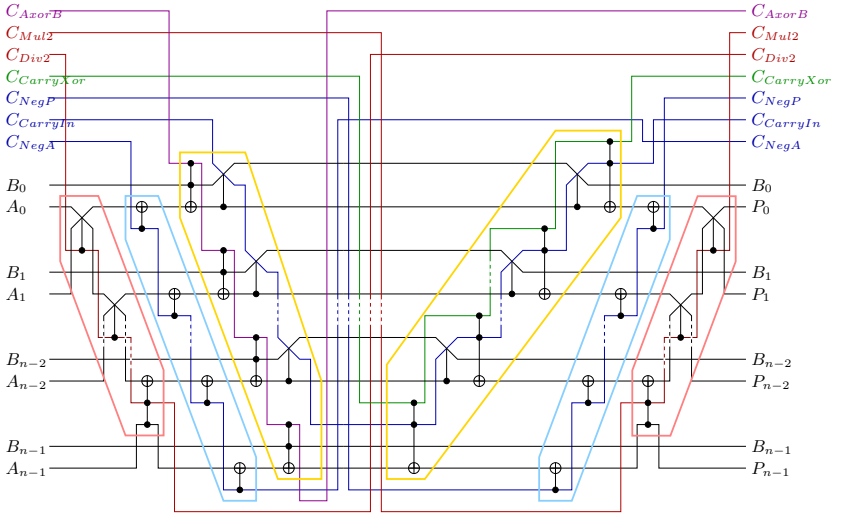

Fig. 5. Logic design of the extended reversible ALU. The division and multiplication by 2 are shown (in red boxes) furthest to the left and right, respectably.

4.1 Reversible Arithmetic-Logic Unit, ALU

The Arithmetic Logic Unit (ALU) is a central part of the processor. In a conventional ALU design all possible arithmetic-logic operations are computed in parallel, and afterwards a multiplexer chooses the desired result; all other results are discarded. This is not desirable for a reversible circuit because of the number of resulting garbage bits. An alternative design for reversible ALU has therefore been suggested by the authors [21]. A key element in this ALU design is the V-shaped (forward and backward ripple) reversible binary adder designed by Vedral *et al.* [23] and later improved in [8, 22, 20].

The ALU design follows a strategy that places all logical operations in sequence and then uses controls to ensure that only the desired operation changes the input values. Of the arithmetic-logic instructions in the proposed instruction set, only the division and multiplication by 2 are not supported by this ALU design. Support for these two instructions are added by new forward and backward ripples at each side of the ALU. The forward ripple (division) first rolls one bit from the least to the most significant bit and then uses an exclusive-or to ensure the sign of the two's complement number. The backward ripple (multiplication) is the exact inverse operation. See Fig. 5 for the detailed design.

This sequential ALU design is surprisingly efficient: compared to the reversible ripple-carry adders it has only constant increase in logic depth and a linear increase in gate count. The cost of the ALU in various metrics can be found in Table 3.

Table 3. Costs in various metrics of the extended n-bit ALU compared to an optimized reversible ripple-carry adder and the entire Bob design without memory.

	Reversible n-bit adder [22]	Extended n-bit ALU	Bob architecture without memory
Gate count total	$4n-2$	$8n-4$	473
Feynman gates	$2n$	$2n$	155
Toffoli gates	0	$2n+2$	146
Fredkin gates	$2n-2$	$4n-6$	172
Ancillae bits	1	0	39
Logic depth	$3n-1$	$3n+3$	
Logic width	$2n+1$	$2n+7$	

4.2 Address Calculation

The address calculation depends both on the semantics of the overall architecture and the instruction set: the architecture specifies in which order to update the special purpose registers (*pc*, *br*, *dir*), while the choice of branch instructions determines the register updates. Adapting the previously described semantics for reversible control (Sect. 2), the address calculation in Bob has the following steps, cf. Fig. 4.

1. *Branch check.* We check if the current instruction is a branch instruction; in the case of a conditional branch instruction we also check if the condition is satisfied. If both evaluate to true, a *doBranch* signal to update the branch register is sent.
2. *Swapping branch register.* Now, we decide what to update. Often, it will be the value in the branch register, but if the current instruction is a swap-branch-register instruction (`SWB` or `RSWB`), then we must update the value of the given general purpose register instead. To do this we use a 2:2 reversible multiplexer (implemented using an array of Fredkin gates), where a control line decides if the inputs are swapped.
3. *Updating the branch register.* If *doBranch* is TRUE then the value of the branch register is updated with the value of the offset. The offset is added if the direction bit is FALSE (forward execution), otherwise subtracted if the direction bit is TRUE (backwards execution), using a simplified ALU.
4. *Updating the direction bit.* In case of an `RSWB` instruction we must invert the direction bit; this is done with a controlled-not gate.
5. *Updating the program counter.* If the updated branch offset equals 0, then the program counter is updated with 1 to step one instruction ahead. Otherwise the program counter is updated with the value of the branch register. Again, the update is either addition or subtraction depending on the direction bit and implemented with a simplified ALU.

```
module alu
  (input  [15:0] A, B
  ,input  C_negA,   C_carryIn,   C_AxorB,   C_carryXor,   C_negP,   C_div2,   C_mul2
  ,output [15:0] P, B_o
  ,output C_negA_o, C_carryIn_o, C_AxorB_o, C_carryXor_o, C_negP_o, C_div2_o, C_mul2_o
  );
    wire [15:0] tmp1, tmp2, tmp3, tmp4, tmp5;
  // DIV2
    assign tmp1[13:0] = (C_div2 ? A[14:1] : A[13:0]);
    assign tmp1[15]   = A[15];
    assign tmp1[14]   = (C_div2 ? A[15] ^ A[0] : A[14]);
  // ADD, SUB, NEG, XOR
    assign tmp2  = (C_negA ? ~tmp1    : tmp1);
    assign tmp3  = (C_carryIn ? tmp2 + 1 : tmp2);
    assign tmp4  = (C_carryXor ? tmp3 + B : (C_AxorB ? tmp3 ^ B : tmp3));
    assign tmp5  = (C_negP ? ~tmp4    : tmp4);
  // MUL2
    assign P[14:1] = (C_mul2 ? tmp5[13:0] : tmp5[14:1]);
    assign P[15]   = tmp5[15];
    assign P[0]    = (C_mul2 ? tmp5[15] ^ tmp5[14] : tmp5[0]);
endmodule
```

Fig. 6. Verilog module for the ALU. Assignments to unchanged outwires (denoted by "*wirename*_o") have been removed for brevity.

6. *Inverse branch check.* To perform the address calculation more efficiently some temporary control values are used, and the final step is to uncompute these. For this, we use the exact inverse of the branch check, explained in the first step.

Previous designs [12,24], which use an *unconditional-branch-and-reverse* instruction to reverse the execution direction, cf. [4], requires two adders in the update of the branch register, compared to one adder for our design.

4.3 Verification

To test the correctness of the design, a Verilog program was implemented and simulated using ModelSim. This language and tool has no built-in support for reversible circuits, but by imposing the Verilog program with the restriction of only using reversible updates, this simulation verifies the correctness of the design. As an example, Fig. 6 shows the implementation of the ALU module. The entire Bob implementation is about 800 lines of pretty-printed code and uses 20 modules. We will not report on timing and other results from this simulation, as these do not yield any additional insights into the design of the architecture.

A future implementation using a *reversible* specification language, such as SyReC [26], is desirable. The effect on the cost of such an implementation compared to custom design of Bob (see Table 3) is hard to predict and depends on the abstraction level of the implementation.

5 Conclusion

We have presented the design of a purely reversible computing architecture with a novel and efficient address calculation and a small, but expressive instruction set containing 17 locally-invertible instructions.[3] The instruction set is r-Turing complete and well-suited as the target language of a compiler from existing high-level structured reversible programming languages. The logical design uses in total only 473 reversible gates (see Table 3), which amounts to 6328 transistors in the *adiabatic dual-line pass-transister* technology [9].

This demonstrates that the design of a complete reversible computing architecture, as presented in this paper, can serve as the core of a simple programmable reversible computing system. Even though our reversible computing architecture does not offer the advanced and sophisticated features of mainstream general-purpose computers, the simplicity makes our design suited as part of special-purpose embedded systems that works without user interaction.

Clearly, further work is required both on the hardware side (including the design and synthesis of reversible circuits with different technologies), as well as on the software side, for fully reaping the low-power benefits of reversible computing systems. This is especially true for the implementation or interfacing of memory.

References

1. Axelsen, H.B.: Clean Translation of an Imperative Reversible Programming Language. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 144–163. Springer, Heidelberg (2011)
2. Axelsen, H.B., Glück, R.: A Simple and Efficient Universal Reversible Turing Machine. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 117–128. Springer, Heidelberg (2011)
3. Axelsen, H.B., Glück, R.: What Do Reversible Programs Compute? In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011)
4. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible Machine Code and Its Abstract Processor Architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
5. Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D.P., Margolus, N., Shor, P., Sleator, T., Smolin, J.A., Weinfurter, H.: Elementary gates for quantum computation. Physical Review A 52(5), 3457–3467 (1995)
6. Burignat, S., Thomsen, M.K., Klimczak, M., Olczak, M., De Vos, A.: Interfacing Reversible Pass-Transistor CMOS Chips with Conventional Restoring CMOS Circuits. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7156, pp. 113–123. Springer, Heidelberg (2012)
7. Cezzar, R.: The design of a processor architecture capable of forward and reverse execution. In: IEEE Proceedings of the SOUTHEASTCON 1991, vol. 2, pp. 885–890. IEEE (1991)

[3] As a historical remark, EDSAC in 1949 had about the same number of irreversible instructions.

8. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit. arXiv:quant-ph/0410184v1 (2005)
9. De Vos, A.: Reversible Computing: Fundamentals, Quantum Computing and Applications. Wiley-VCH (2010)
10. De Vos, A., Burignat, S., Thomsen, M.K.: Reversible implementation of a discrete integer linear transformation. Journal of Multiple-Valued Logic and Soft Computing 18(1), 25–35 (2012)
11. Feynman, R.P.: Feynman Lectures on Computation. Addison-Wesley (1996)
12. Frank, M.P.: Reversibility for Efficient Computing. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (1999)
13. Fredkin, E., Toffoli, T.: Conservative logic. International Journal of Theoretical Physics 21(3-4), 219–253 (1982)
14. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5(3), 183–191 (1961)
15. Lutz, C.: Janus: A time-reversible language. A letter to R. Landauer (1986), `http://www.tetsuo.jp/ref/janus.html`
16. Maslov, D., Dueck, G., Miller, D.: Synthesis of Fredkin-Toffoli reversible networks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13(6), 765–769 (2005)
17. Morita, K.: A Simple Universal Logic Element and Cellular Automata for Reversible Computing. In: Margenstern, M., Rogozhin, Y. (eds.) MCU 2001. LNCS, vol. 2055, pp. 102–113. Springer, Heidelberg (2001)
18. Patterson, D.A., Hennessy, J.L.: Computer Organization & Design: the hardware/software interface, 2nd edn. Morgan Kaufmann Publishers (1997)
19. Shende, V., Bullock, S., Markov, I.: Synthesis of quantum-logic circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25(6), 1000–1010 (2006)
20. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(1), 205–222 (2009)
21. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. Journal of Physics A: Mathematical and Theoretical 43(38), 382002 (2010)
22. Van Rentergem, Y., De Vos, A.: Optimal design of a reversible full adder. International Journal of Unconventional Computing 1(4), 339–355 (2005)
23. Vedral, V., Barenco, A., Ekert, A.: Quantum networks for elementary arithmetic operations. Physical Review A 54(1), 147–153 (1996)
24. Vieri, C.J.: Reversible Computer Engineering and Architecture. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (1999)
25. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer Science (2010)
26. Wille, R., Offermann, S., Drechsler, R.: SyReC: A programming language for synthesis of reversible circuits. In: Proceedings of the Forum on Specification & Design Languages, pp. 1–6. IET, Southhampton (2010)
27. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM (2008)
28. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of Partial Evaluation and Program Manipulation, pp. 144–153. ACM (2007)