DEPARTMENT OF COMPUTER SCIENCE (DIKU)
UNIVERSITY OF COPENHAGEN

Design of Reversible Logic Circuits using Standard Cells

– Standard Cells *and* Functional Programming

Michael Kirkedal Thomsen

Technical Report no. 2012-03

ISSN: 0107-8283

Abstract

This technical report shows the design and layout of a library of three reversible logic gates designed with the standard cell methodology. The reversible gates are based on complementary pass-transistor logic and have been validated with simulations, a layout vs. schematic check, and a design rule check. The standard cells have been used in the design and layout of a novel 4-bit reversible arithmetic logic unit. After validation this ALU has been fabricated and packaged in a DIL48 chip.

The standard cell gate library described here is a first investigation towards a computer aided design flow for reversible logic that includes cell placement and routing. The connection between the standard cells and a combinator-based reversible functional languages is described.

Keywords: Reversible computing, reversible circuits, standard cells, CMOS, computer aided design

Contents

1 Introduction

Reversible computation [16, 5] is a research area characterized by having only computational models that are both forward and backward deterministic. The motivation for using these models comes from the prospect of removing the energy dissipation that is caused by information destruction. Though recent experimental results have confirmed Landauer's theory [6], the results of applying the model to devices in today's computing technologies is still unknown.

We will here look at the model of reversible logic and logic circuits (as explained in Sec. 1.1) and investigate how we can implement circuits with *computer aided design* (CAD). In Sec. 2 we discuss how we can make reversible logic circuits in CMOS, and discuss the benefits and drawbacks of the different logic families. In Sec. 3 we use one of these logic families to implement the basic reversible gates in "first approach" standard cells (a design methodology from the static CMOS family that is very suitable for CAD). In the end of the section (Sec. 3.5) we conclude and suggest future improvements. To show that these standard cells actually work, we have implemented and fabricated a reversible arithmetic logic unit (Sec. 4). Finally, in Sec. 5, we discuss how a recently developed reversible functional language can be used to aid the CAD process, and, in the future, make it possible to design more complex reversible circuits. We shall discuss related work throughout this report.

1.1 Reversible Logic

To describe *reversible logic circuits*, we use the formalism of Toffoli, Fredkin [31, 14] and Barenco *et al.* [4]. That is, a *reversible gate* is defined as a bijective boolean function from n to n values. There exist many of these gates, but we restrict ourselves to the following basic reversible logic gates [4]:

- The Not gate (Fig. 1); the only gate from conventional logic that is reversible.
- The *Feynman gate* (Feyn, Fig. 2), or controlled-not gate, negates the input A iff the control C is true.
- The *Toffoli gate* (Toff, Fig. 3), or controlled-controlled-not gate, negates the input A iff both controls C_1 and C_2 are true.
- The *Fredkin gate* (Fred, Fig. 4), or controlled-swap gate, swaps the two inputs A and B iff the control C is true.

A *reversible circuit* is an acyclic network of reversible gates, where *fan-out* is not permitted.

In this work we take a completely *clean approach* [2]. This makes the number of auxiliary bits used an important non-standard characteristic of reversible

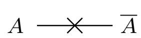

Figure 1: Not gate

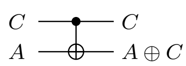

Figure 2: Feynman gate, Feyn.

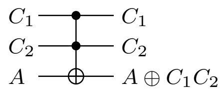

Figure 3: Toffoli gate, Toff.

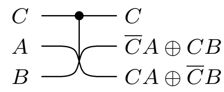

Figure 4: Fredkin gate, Fred.

circuits. We define a *garbage* bit as a non-constant output line that is not part of the desired result but is required for logical reversibility, An *ancilla* bit is a bit-line that is assured to be constant at both input and output. Being *clean* means that no garbage is allowed, as garbage bits accumulate over repeated computation (which is likely to lead to information destruction). We can, however, compute temporary values if these are uncomputed again at a later time; we then will call the total usage ancillae, which can be reused with each new computation of the circuit.

2 Implementation of Reversible Gates

Reversible computation is related to other emerging technologies such as quantum computation [12, 23, 7], optical computing [9], and nanotechnologies [22] that use a similar or slightly extended set of gates.

First implementations and fabrications of reversible logic in CMOS technology have also been accomplished (e.g. [24]). These exploit that reversible logic is particularly suitable

- when it comes to reuse of signal energy (in contrast to static CMOS logic that sinks the signal energy with each gate), and,
- when using *adiabatic switching* [15, 1] to switch transistors in a more energy efficient way.

In fact, SPICE simulations of reversible circuits have shown that such implementations have the potential to reduce energy consumption by a factor of 10 [10, 11].

A drawback of these implementations comes from another law related to transistors, namely that the energy consumption is directly related to the execution frequency. If one performs many computations every second, the energy consumption per computation rises. Performing fewer computations lowers the energy consumption per computation.

Of course, this implies that not all applications are necessarily suited for implementation using reversible circuits. However, many embedded devices do not need to perform billions of computations every second.

In the rest of this section will focus on how to implement reversible gates in CMOS. First, we briefly review some basics of CMOS transistor implementation [21, 35] as used in this work, and afterward we explain how this is used in an implementation of reversible gates.

2.1 Basic Transistor Theory

When either an nMOS or a pMOS transistor is used alone as a switch they are referred to as a *pass-transistor*, but neither of them are perfect switches. nMOS

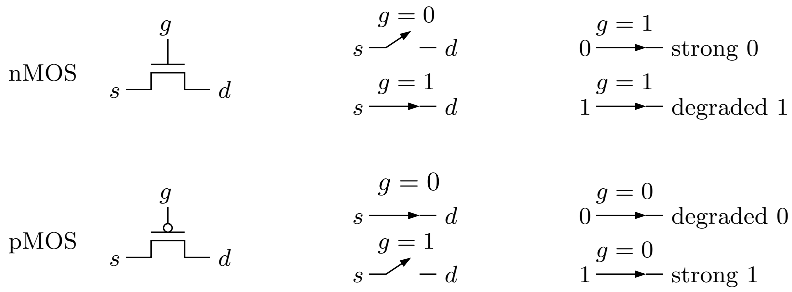

Figure 5: nMOS and pMOS transistor description. Figure adapted from [35].

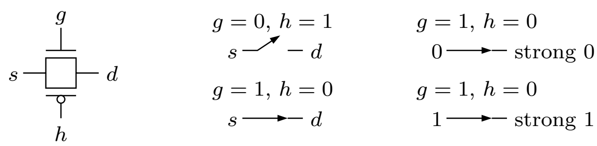

Figure 6: Pass-gate description. Figure adapted from [35].

transistors are almost perfect (called *strong*) for passing a low-voltage signal (FALSE) between its source (s) and drain (d), but very bad (called *degraded* or *weak*) for passing a high-voltage signal (TRUE). pMOS transistors on the other hand pass a degraded low-voltage and a strong high-voltage (see Fig. 5).

As a solution we can use a nMOS and a pMOS transistor in parallel to make a gate that passes both a strong low-voltage and a strong high-voltage signal (see Fig. 6). This gate is called a *pass gate* or *transmission gate*. The two gate signals can be used independently, but when designing circuits with pass gates we often have that one gate signals g, have the negated value of the other signal h ($h = \overline{g}$). We, therefore, have two complementary lines for all signals (g and $\overline{g}$) and, thus, call this *complementary pass-transistor logic* (CPL) or *dual-line* pass-transistor logic.

2.2 Adiabatic Switching

Adiabatic switching [15] was introduced as a way to reduce the dissipation caused by transistor switching and to reuse the signal energy. The word *adiabatic* comes from physics and, for transistors, it is used to describe that the energy dissipated for transistor switching tends towards zero when the switching time tends towards infinity. The only way that we can increase the switching time of the transistors is by using a control signal, g, that changes gradually from no signal to either TRUE or FALSE instead of a signal that changes abruptly.

Transistor theory provides the two following rules that adiabatic circuits must follow [13]:

- The control signal of a transistor must never be set (TRUE or FALSE) when there is a significant *voltage* difference between its source and drain.

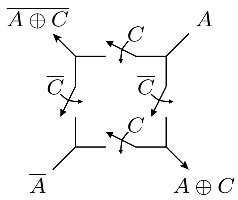

Figure 7: Implementation of Feynman gate in R-CPL using 4 pass gates (8 transistors). Figure from [32]

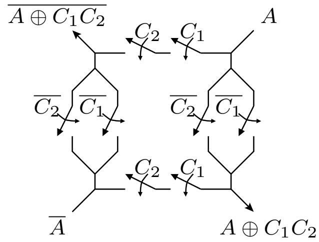

Figure 8: Implementation of Toffoli gate in R-CPL using 8 pass gates (16 transistors). Figure from [32]

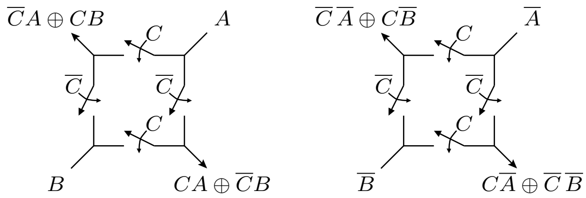

Figure 9: Implementation of Fredkin gate in R-CPL using 8 pass gates (16 transistors). Figure from [32]

- The control signal of a transistor must never be turned off when there is a significant *current* flowing between its source and drain.

Notice that an adiabatic circuit is *not* necessarily a reversible circuit and *vice versa.* In the following we will describe an adiabatic logic family that implements the reversible gates.

2.3 Reversible Complementary Pass-Transistor Logic

Pass-transistor logic has been used in conventional computers for many years in order to improve specialized circuits such as *Static RAM* and other circuits that use many XORs[1]. This was used for reversible gates by De Vos [10], who with the help of Van Rentergem implemented the reversible gates [32] from Sect. 1.1. This logic family is called *reversible complementary pass-transistor logic* (R-CPL).

The gates are designed as controlled rings, where pass gates are used to open or close connections according to the desired logical operation (Figs. 7, 8, and 9). As we can see there is no V_{dd} or V_{ss} in these designs, implying that no charge can be added (or removed) and thus the gates must be *parity preserving*[2]. The Cnot and the Not gates are not *logically* parity preserving in the sense of conservative logic, but we can make them so, by using a complementary-line

[1] Static CMOS is poorly suited for the implementation of XOR gates compared to AND and OR gates

[2] A gate or circuit is parity preserving if the number of input and output lines that has the value TRUE are equal.

implementation[3]. The Fred gate is parity preserving which we can see in the gate implementation as two non-connected cycles. Furthermore the design of the gates implies that current can flow both ways though the circuits and, thus, the inverse of a gate is itself with the input lines swapped with the output lines.

3 Standard Cells for Reversible Logic

The *standard cell methodology* is the most widely used way to design digital ASICs today. The idea is to make automated designs much simpler by having a *standard cell library* where the elements easily fits together. Each cell (or gate) in this library implements a logical operation (NOT, AND, OR, *etc.*) and must uphold some constraints; *e.g.* they must have the same physical height and some wire connections must match. On the other hand, then the functional complexity of the gates are much different, so their standard cell implementation can vary in, *e.g.*, physical width and number of transistors.

3.1 Designing the Standard Cells

The standard cells made in this work implement the basic reversible gates: Feynman, Toffoli, and Fredkin gates. No layout for the Not gate is made as this gate, in a dual-line technology, is a simple swap of the wires; no transistors are needed.

The idea is to design the standard cells such that they mirror the diagram notation in that all signals flow from left to right (or opposite for the inverse circuit). By definition a reversible gate has the same number of inputs and outputs, and because of the no fan-out restriction, routing between the cells is simple and placing the cells directly side-by-side is possible. This will not work for static CMOS, which have many-to-one gates and fan-out. Our standard cells will have the following properties:

- All basic reversible gates are either two- or three-input gates and, therefore, on each side there is up to six input/output pins (three dual-line pins).
- A V_{dd} and a V_{ss}-rail are added on the top and bottom, respectively. These are important for polarization of the substrate and the well.
- Only two metal layers are used. This will leave enough metal layers for routing between the cells.
- The height of all gates is 15 μm. An n-well spanning the entire width has a height of 8 μm. To ease hand-designing, each of the six pins are 1 μm high and are equally spaced with a 1 μm gap. In addition to this, the two rails gives a total height of 15 μm. The height of the n-well has been chosen to fit the two rows of transistors and is larger than half the height of the cell because p-transistors must be about three times wider than n-transistors to have similar resistance.

[3]Any gate can be made parity preserving by adding a complementary line, implying that parity preserving gates are not necessarily reversible gates and *vice versa*.

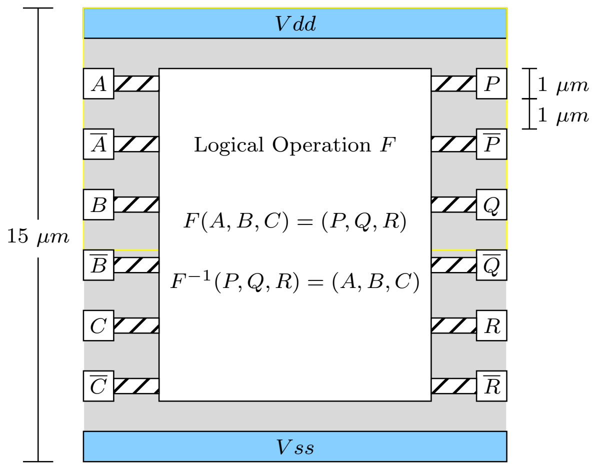

Figure 10: General layout of the basic cells. The height is fixed but the width can vary. The yellow box is the n-well.

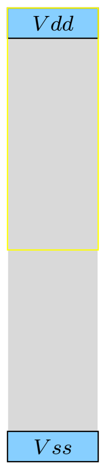

Figure 11: Layout of spacing cells that match the gate cells.

- All pins to and from the cell are made in metal layer 1. Routing inside the cell is made through both metal layers 1 and 2. (The metal layers are numbered from 1 and up starting from the layer above the polysilicon and diffusion. In our designs we use at most three metal layers.)

Some "space" cells of different width are made, so the designer can use them when more space for wiring is needed. These only contains V_{dd}, V_{ss}, and the well. A general layout of standard cells is shown in Fig. 10 and an example space cell is shown in Fig. 11. The (up to three) inputs are here labeled A, B, and C, with the outputs labeled P, Q, and R.

All layouts are made in 0.35 μm (transistor length) technology from *ON Semiconductor* using the *Cadence Virtuoso*© CAD tool. It is based on a p-substrate where an n-well is added. All layouts have successfully been validated by the *design rule check* from the foundry and a *layout versus schematic* check have successfully validated the functionality with respect to the schematics from Sec. 2. Previously, the schematics were validated by electrical simulations using the Spectre© simulator that is part of the Cadence tool.

3.2 The Feynman Gate

The simplest of the cells are the Feynman gate; it has a width of 10.5 μm, and uses 8 transistors. The first step in the design is to find a good geometric placement of the transistors. We want this placement to have the smallest width possible (to reduce the total circuit area), but at the same time we also want the routing within the gate to be simple; we shall not use more than metal layers 1 and 2.

The abstract layout of the Feynman gate is shown in Fig. 12. In the transistor-placement each column contains one n- and one p-transistor, both connected with the same source and drain signal and this will, thus, work as a pass-transistor. There are four of these pass-transistors, which fits the schematic

	Feynman gate	Toffoli gate	Fredkin gate
Inputs/outputs	2	3	3
Number of transistors	8	16	16
Cell width	10.5 μm	20 μm	16 μm
- hereof for routing only	2.5 μm	7 μm	5 μm

Table 1: Summary of the sizes of the cells

from Sec. 2. The advantage of this abstract layout is that all source and drain connections are easy: Q, $\overline{Q}$, B, and $\overline{B}$ are all placed on a single vertical metal wire. The drawback is that the transistor routing is not simple: notice that the pins to the gates, A and $\overline{A}$, is alternate in both the vertical and horizontal direction.

The actual CMOS layout of the abstract layout is shown in Fig. 14. Before explaining the layout (in Sec. 3.2.1), we would like to draw attention to the legend in Fig. 13. The top of the legend shows the color scheme used for polysilicon, diffusion (both n+ and p+), and the two used metal layers. The slightly larger boxes in the middle defines the area for the n-well and how to specify that diffusion is either of p+ or n+ type; these two basically determines if the transistors are n- or p-transistors. Finally, the bottom of the legend shows a via between metal layers 1 and 2, and the contacts (connections to polysilicon and diffusion); the vias can be hard to locate in the layout as they often are placed on top of the contacts: but in general, they are placed at the ends of metal 2 wires.

3.2.1 Design Rules and Layout Details

The standard cells designed here are intended for use in actual chips and, therefore, the fabrication process imposes some restrictions on the designed layout. These restrictions ensures that there is a high probability that circuit functions correctly (allthrough it is not guaranteed). Most of the restrictions comes from the making of the lithography masks.

In the design phase it is easy violate one or more of these rules, so the design tools provide a *design rule check* (DRC) that can validate the design agains a list of rules provided by the foundry. The most well-known of these rules is the (minimal) transistor length (the single number that characterizes the technology: 0.35 μm for this particular technology), which manifests as the width of the polysilicon when it intersects the diffusion in the layout.

Most of the design rules are very technical: they defines the minimal width of wires, spacing between wires, the amount of metal that is needed around vias and contacts, *etc.* However, all these rules do *not* necessary reflect the best design choices. For example, in our designs the nMOS transistors are three times wider than the pMOS transistors (which is equal to the minimum transistor width), because we would like to have similar resistance in the two types of transistors.

The layout of the Feynman gate is shown in Fig. 14 and it upholds the design rules of the technology. Table 1 lists some basic facts about layouts of the three reversible gates.

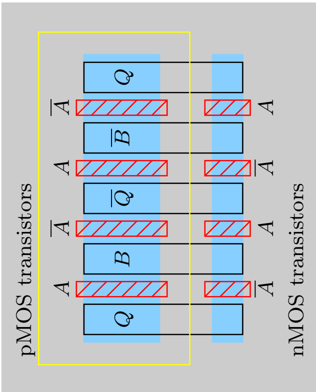

Figure 12: Abstract layout of the Feynman gate.

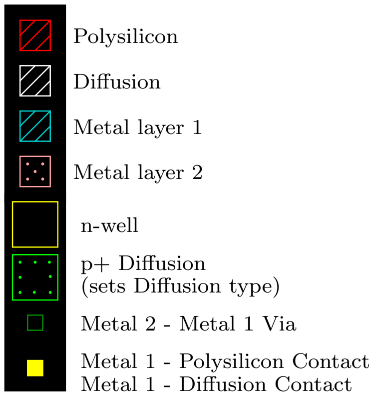

Figure 13: Legend for CMOS layouts. All diffusion that is within the dotted green boxes are p+ diffusion, while the rest is n+ diffusion.

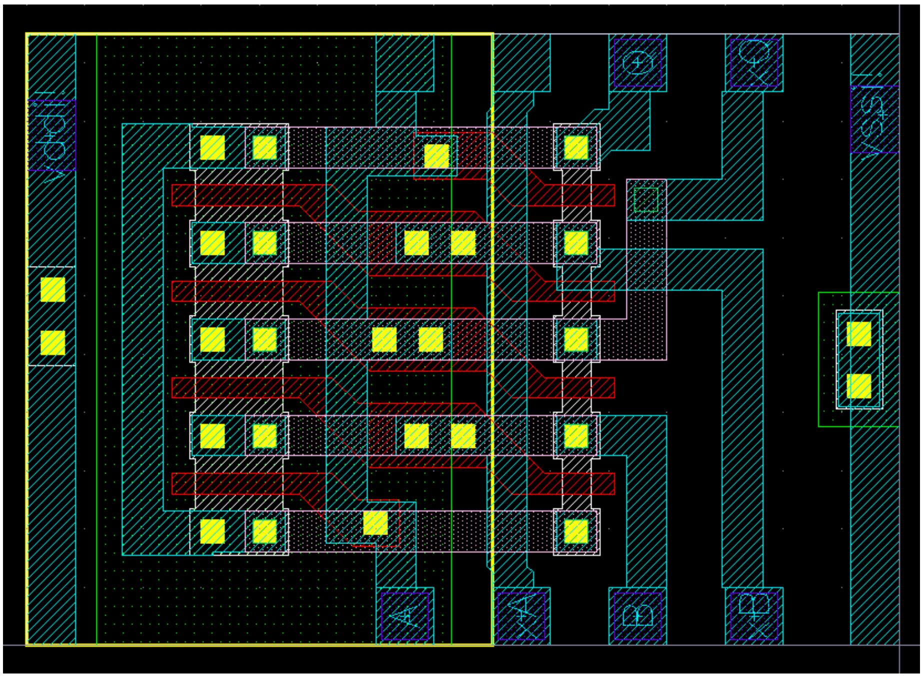

Figure 14: CMOS layout of the Feynman gate.

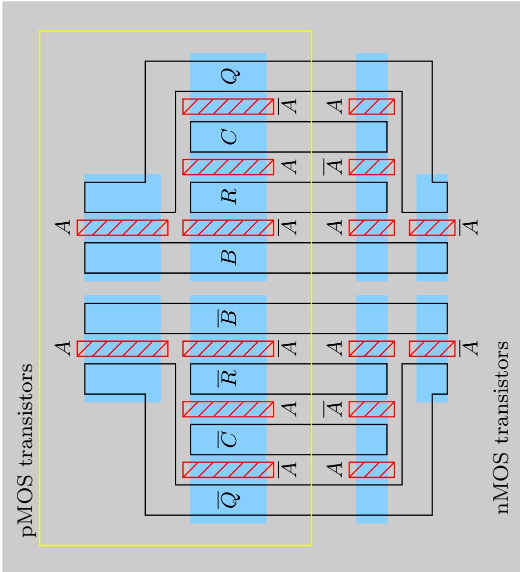

Figure 15: Abstract layout of the Fredkin gate.

3.3 The Fredkin Gate

Functionally, the Fredkin gate seems more complicated than the Feynman gate. It has one extra input (three in total) and updates two outputs. Also, the logical formulation of a swap is more complex (see Fig. 4). But the schematic in Fig. 9 shows that the actual CPL-implementation is equal to two Feynman gates, where one gate swaps the values (A and B) and the other gate swaps the complemented values ($\overline{A}$ and $\overline{B}$) both depending on the control (C and $\overline{C}$). That the values and complemented values can be calculated independently is not a complete surprise. The Fredkin gate is parity preserving and the trick of using dual-line values is, thus, not necessary, but we still have and compute to both complementary lines such that they can be used in the next gate.

The layout of the Fredkin gate does not, however, precisely mirror a double-Feynman gate (see Fig. 15 for an abstract layout). Instead of having 2×4 transistors on a single row, the transistors have been arranged in two rows by moving one transistor of each type. This reduces the necessary area usage but also makes routing of the wires more complicated. The layout of the gate is shown in Fig. 16. The width of the gate is 20 μm, so compared with the Feynman gate (that has a width of 10.5 μm) the area reduction is not impressive, which is actually due to the more complicated routing. At each side of the Fredkin gate about 3 μm is needed for inputs/outputs and to route the wires, which can be compared to the about 1.5 μm for the Feynman gate.

In total only 12 μm of the 20 μm wide cell is used for transistors. It is expected that the reversible gates will use more 'real estage' compared to a static CMOS gate in order to implement a similar functionality, and using only half of the area for transistors does not help the area overhead. In Sec. 3.5 we will discuss how this might be improved.

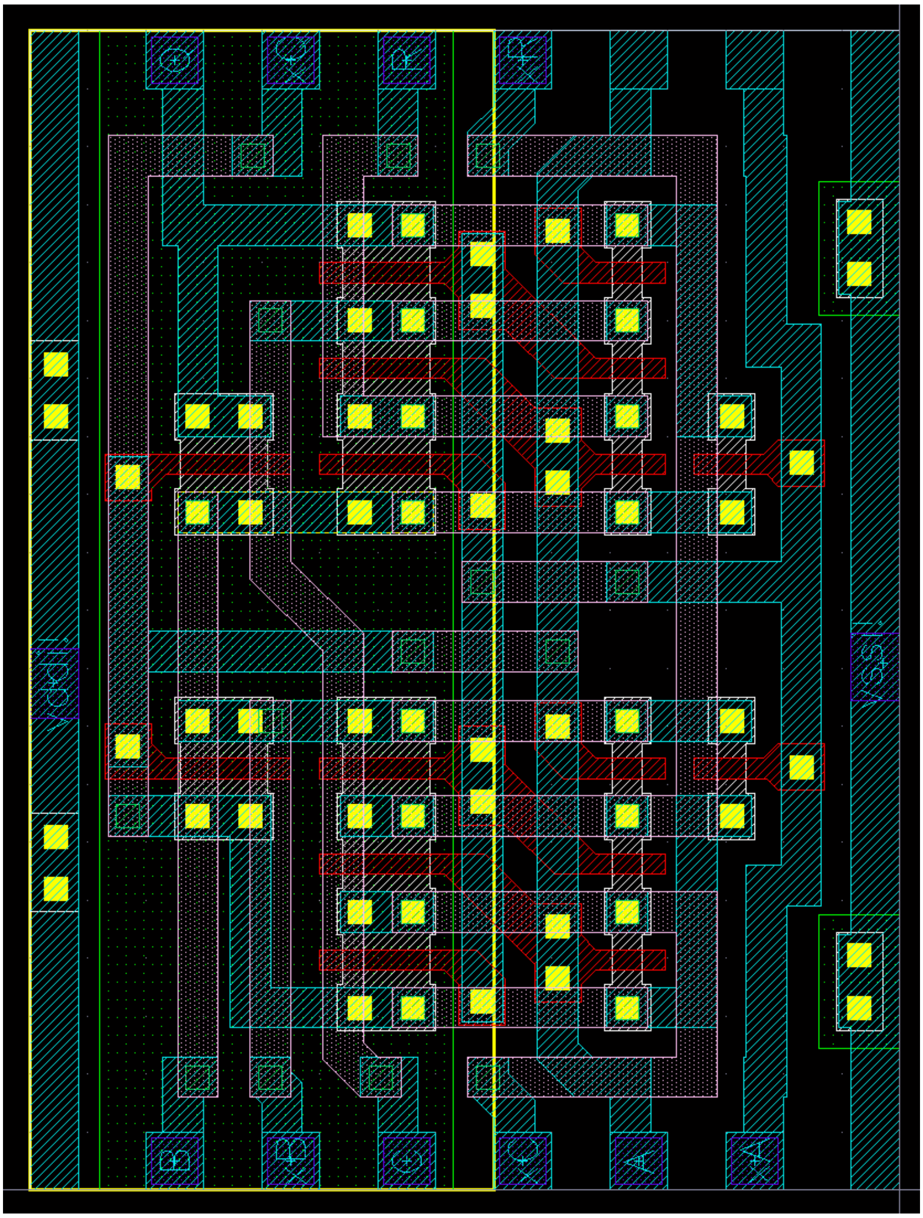

Figure 16: CMOS layout of the Fredkin gate.

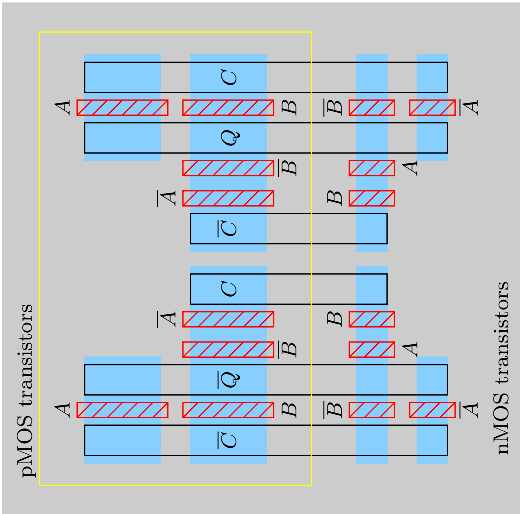

Figure 17: Abstract layout of the Toffoli gate.

3.4 The Toffoli Gate

In contrast to the Fredkin gate, the schematics of the Toffoli gate (Fig. 8) shows that this gate is more complicated than the Feynman gate. Twice, it contains two pass-gates in parallel and two pass-gates in serial. This is obvious in the abstract layout (Fig. 17), where, for example, the two parallel pMOS transistors are shown on the left and right side (which uses two rows) and the serial pMOS transistors are shown in the middle. Notice, that it is possible to put two wires of polysilicon between the diffusion and, thus, reduce the total circuit area.

The layout of the Toffoli gate is shown in Fig. 18 and is only 16 μm wide. This is actually a very compact design, although, like the Fredkin gate, each side of the cell adds almost 3 μm to route the pins.

3.5 Towards Computer Aided Design

The standard cells presented here were designed for use in “hand-made” layouts, but are still based on ideas from CAD methods. Work with the cells has, however, shown that it is possible to further improve the layouts. In this section we will discuss some future design approaches.

3.5.1 Number of Gates

The general design idea was to mirror the diagram notation in the gate layouts, such that the inputs are on one side and the outputs in the other side. All signals would then flow from the left to the right. A plan, which followed directly from the diagrams, was that extra gates should be designed where the inputs (and outputs) were permuted and/or negated. This would result in a large set of gates, but would make routing much easier: In many cases it could just be placing one gate beside the next.

The problem is that actual logic implementations [28, 30] have shown that this rarely occurs. Often, one of the outputs is used with other signals (*e.g.* in

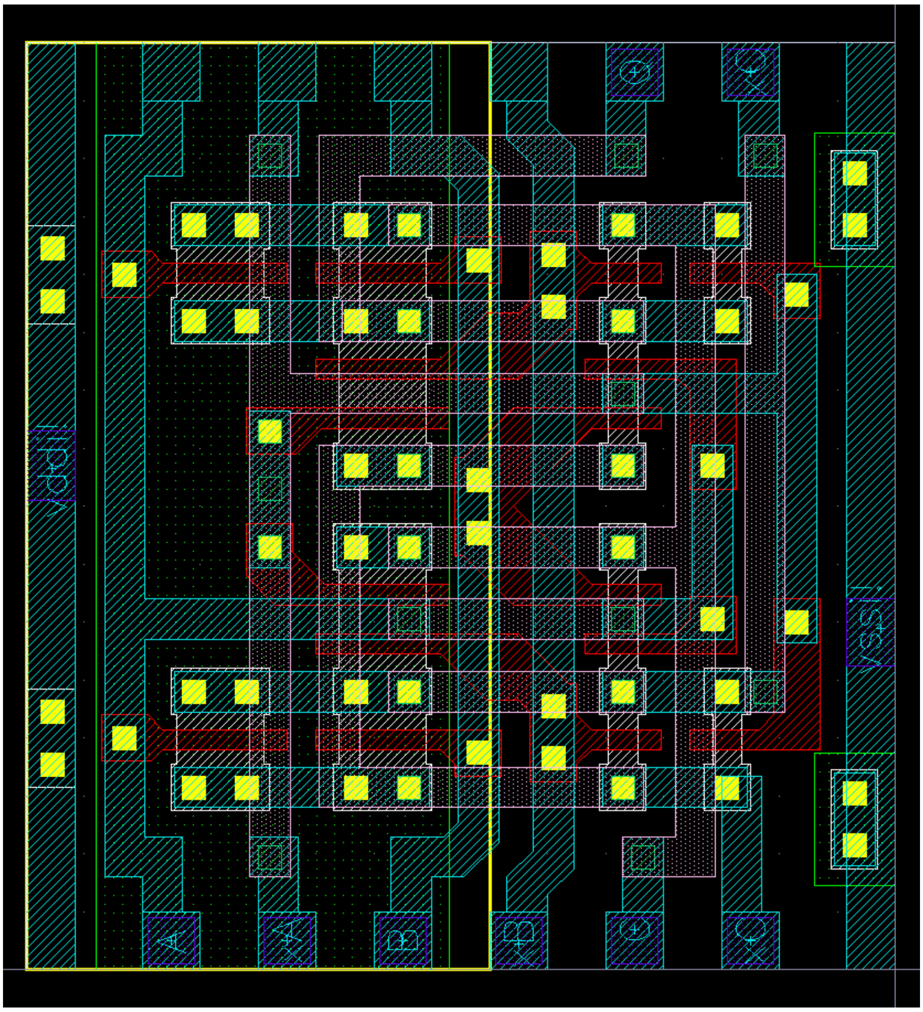

Figure 18: CMOS layout of the Toffoli gate.

a ripple) and space between the cells is, thus, needed.

Also, CMOS technology develop so fast that a new technologi is introduced every 2 to 4 years and each time all the gates must be redrawn in this new technology. Each foundry even has its own design rules, so changing from one foundry to another (using the same transistor size) is likely to incur some changes to the gate layouts. One logic family of asynchronous logic also uses complementary dual-lines and here they use this to their advantage by implementing all 2- and 3-input gates [18, 17], which is only two 2-input gates (logic-and and xor) and also very few 3-input gates. The rest of the gates can be implemented by negating inputs and/or outputs, which is a simple line swap in a dual-line technology.

Taking this strategy even further, it is not even necessary to have an implementation of the Fredkin gate, as we know that it is implemented with two Feynman gates. The exact number of needed gates should be investigated.

3.5.2 Input/Output Placement and Gate Size

A practical experience from designing the gates as similar to the diagrams, was also that the cells use a lot of overhead space. Both the Fredkin and the Toffoli gate have on each side between 2.5 μm and 3 μm of wiring before the transistors are placed. The best way to remove this is to follow the strategy from static CMOS standard cell and place the pins inside the cells.

This brings us directly to a second problem, namely that all pins are placed in metal layer 1. The reversible gates are functionally more advanced than conventional logic gates, which is shown by the use of metal layer 2 for routing inside the standard cell; something that is not much used in static CMOS gates. All pins should instead be placed in metal layer 2 (or perhaps 3) and metal layer 1 (and perhaps also metal 2) should not be used for routing between the gates at all (only inside the gate). This will reduce the number of metal layers that can be used for automatic routing, but will not cause any problem, as modern chips have at least 9 metal layers, and routing between the reversible gates is expected to be easier than for conventional gates.

3.5.3 Transistor Placement ands Gate Size

In the previous section, we discussed how to reduce the cell area by moving the pins. Another way to reduce the area is to optimize the transistor placement. In the Feynman gate all the transistors are placed on only one row, so we will here look at some alternative designs that can improve this.

The Fredkin gate (and the Toffoli gate) have enough space to fit two rows of transistors (Figs. 16 and 18). And in the Fredkin gate (that is implemented as two Feynman gates) the first step was made by moving one of the transistors. It is, however, possible to place the pMOS and nMOS transistors in a 2 $\times$ 2 grid as shown in Fig. 19 and, thereby, reduce the used area even further. This does, however, make the routing of the gate signals, A and $\overline{A}$, to the transistors more complicated. The biggest problem with this design is perhaps that the transistors connected to Q has been divided into two parts: The pMOS transistors at the top and nMOS transistors at the bottom of the cell.

A solution to this problem is shown in Fig. 20. Here the n-well (and pMOS transistors) have been divided into two parts (at the top and bottom) and the

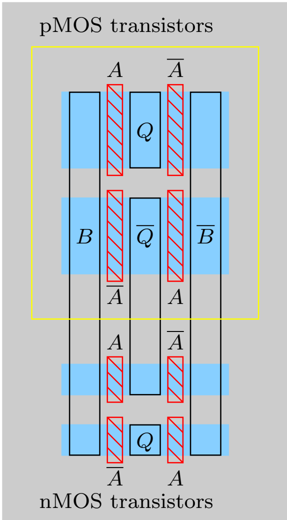

Figure 19: Alternative abstract layout of the Feynman gate.

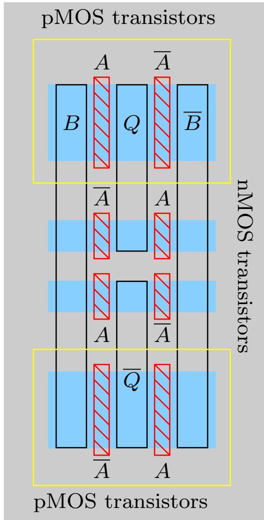

Figure 20: Second alternative abstract layout of the Feynman gate.

nMOS transistors are placed in the middle. Now the transistors connected to both Q and $\overline{Q}$ placed together, while the gate signal to the trasistors (A and $\overline{A}$) have similar routing. The only problem could be that the rail for V_{ss} now must run through the middle of the cell. As these gates have not yet been drawn in actual layout it is unknown if this will work. It is likely that we also have to use metal layer 3 for routing.

4 Implementation and Fabrication of an ALU

In the following we will show how the reversible standard cells have been used to implement a reversible arithmetic logic unit (ALU). This includes pictures of both the schematic and actual layout. The resulting layout have been fabricated and the functionality of the chip has been tested.

The ALU is a central part of a programmable processor [29]; given some control signal, it performs an arithmetic or logical operation on it inputs. In a conventional ALU design the arithmetic-logic operations are all performed in parallel, after which a multiplexer chooses the desired result. All other results are discarded. This is not desirable for a reversible circuit, because of the number of garbage bits this would require.

Instead, the design implements the recent reversible ALU design presented in [30]. This ALU follows a strategy that puts all operations in sequence and then uses the controls to ensure that only the desired operation changes the input. The reversible ALU is based on the V-shaped (forward and backward ripple) reversible binary adder designed by Vedral *et al.* [34] and later improved in [8, 32, 28].

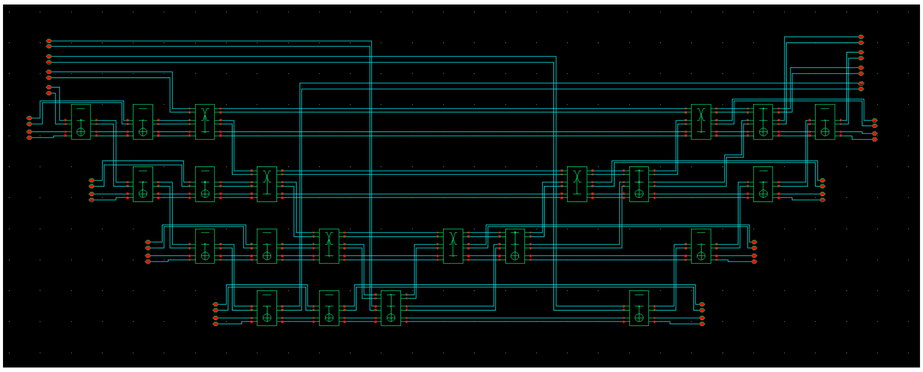

Figure 21: Schematic of the reversible ALU. This schematic is used for simulation and to verify the layout with respect to connections, transistors, and logic gates.

4.1 The Schematic

The ALU design from [30] can have an arbitrary size, but for practical reasons we want the chip to be packaged in a *dual in-line* (DIL) device package with 48 legs, which is the largest DIL package size available. Our implementation has, therefore, been limited to a 4-bit input width. The ALU is implemented using bit-slices so all interesting design choices are shown at this size and there is, therefore, no need to make the implementation more advanced.

A 4-bit input width schematic is shown in Fig. 21. This schematic follows the design from [30] and it is possible, in this figure, to follow the V-shaped forward and backward ripples. The design consists of 12 Feynman gates, 6 Fredkin gates and 4 Toffoli gates.

4.2 The Layout

The ALU has a very regular structure, so it is beneficial to divide the layout into two different types of bit-slices. The first is used for the $n-1$ (3 in this example) least significant bits and implements the entire functionality using six gates. The second bit-slices is only used for the most significant bit and implements a small optimization (using two gates less) that is also used in binary adder design. The layout of the ALU is shown in Fig. 22, where the bit-slice for the most significant bit is to the left. Around the gates (bit-slices) is placed a *power ring* that enables easy and efficient distribution of the V_{dd} and V_{ss}. The shole layout including the power ring has a size of 113 μm × 72 μm. In total this is 8136 μm^2 or about 0.008 mm^2.

The main purpose of the schematic is, in a simple way, to describe the functionality of the circuit, which then is used to verify the layout; also called a *layout vs. chematic* (LVS) check. More specific, each gate (green box) in Fig. 21 is first expanded with its transistor implementation to give a detailed connection diagram. Then transistors are inferred from the layout, and labeled inputs and outputs are matched to verify that the schematic and layout are identical. The gate-schematics also include metrics like transistor width and length These informations are also verified against the layout in the VLS check.

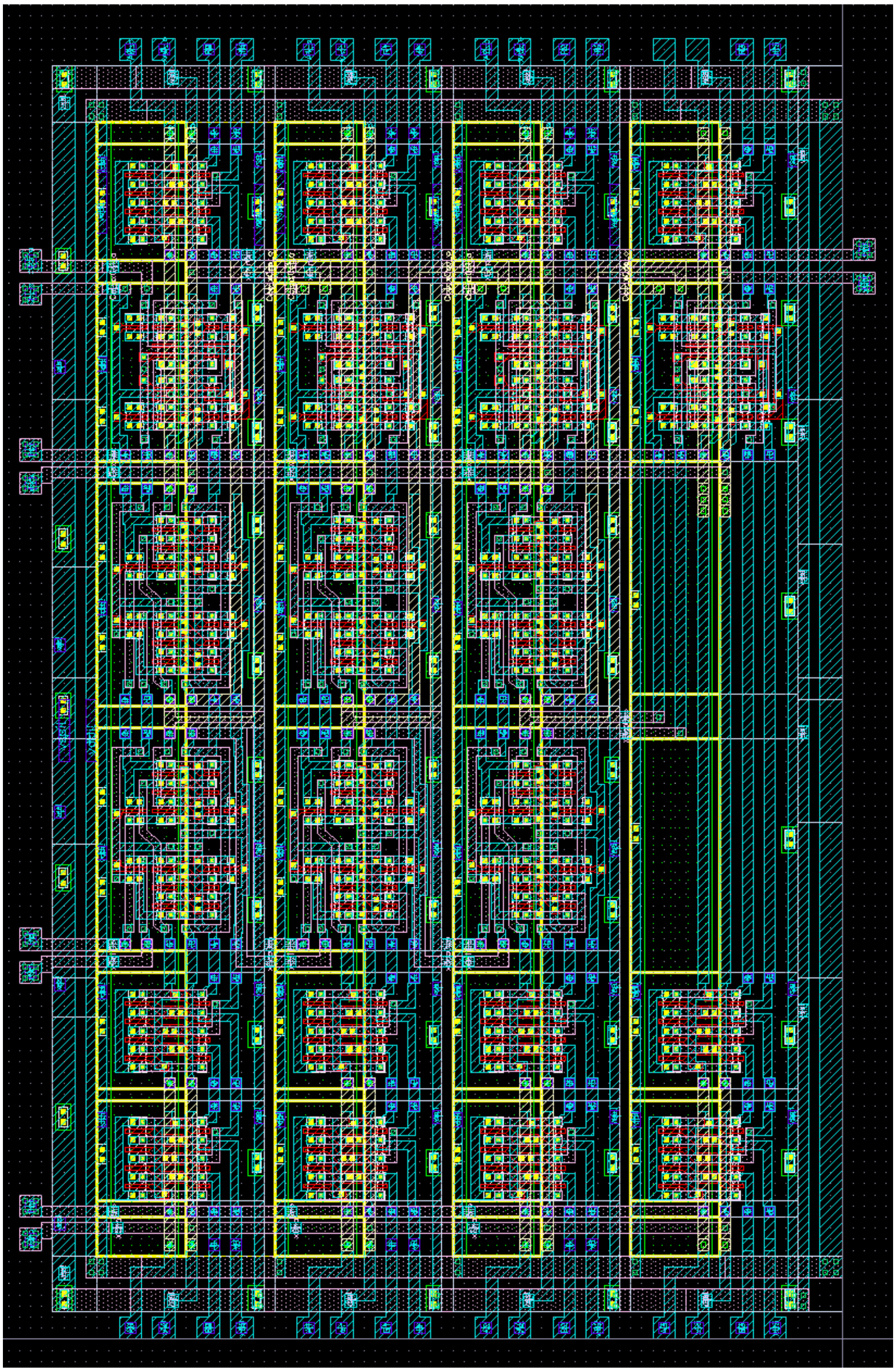

Figure 22: Layout of the reversible 4-bit ALU. Inputs (in the forward direction) are connected at the bottom (with least significant to the left) and the outputs can be read at the top. The five control-lines can be connected at the sides; four at the left side and one at the right side.

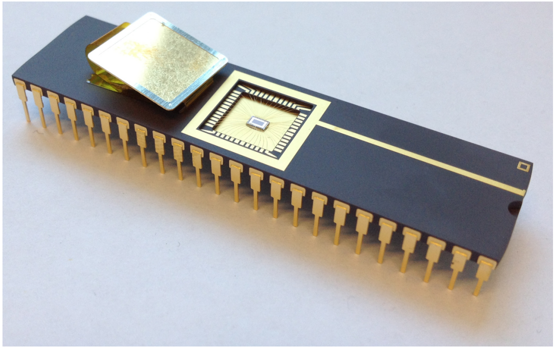

Figure 23: Photograph of the fabricated ALU chip.

4.3 The Pad Ring and Device Package

After the ALU layout has been made, a *pad ring* is added, which is necessary if one will actually use the chip. Its main purpose is to enable easy connection to the device package, but at the same time it also protects the logic circuit against overload by *e.g.* static charge. The ring consists only of predefined elements and defines the physical size of the fabricated circuit. The layout including the pad ring is shown in Fig. 24 where the layout shown in Fig. 22 only fills the small green rectangular in the center of the figure. Also, a picture of the fabricated and packaged circuit is shown in Fig. 23. The whole ALU including the pad ring is 2094.1 μm × 1371,7 μm or 2.87 mm^2; compare this to 0.008 mm^2 for the designed circuit.

4.4 Design Rule Check

The result of the *design rule check* on the layout for the ALU, including the pad ring, (shown in Fig. 25) reveals seven different types of violations; some of them relates to more than hundred places in the layout. The violations are explained below, but none of them cause problems for the fabrication of the circuit.

- **wtopmetal3_aMETAL5** and **...METAL4**: These violations is caused by the pad ring. The chosen layout was defined to have a maximum of 3 metal layers, which was enough for this ALU design, but the pad ring also uses metal layers 4 and 5. The fabrication process supports up to 7 metal layers

- **END_1**: This violation refers to a special “edge of die” box that can be set to improve the result of automatic dummy metal and polysilicon placement. This box is not necessary and, therefore, not added in this layout, thus, resulting in this error.

  Dummy metal and polysilicon is automatically added to the layout to improve the lithography masks and reduce errors in the fabrication process. In modern technologies, where lithography masks are very sensitive to layout changes, advanced algorithms for placing dummy metal is used. In the technology we use this is not the case and more simple approaches, like just adding dummy metal in empty area, are used.

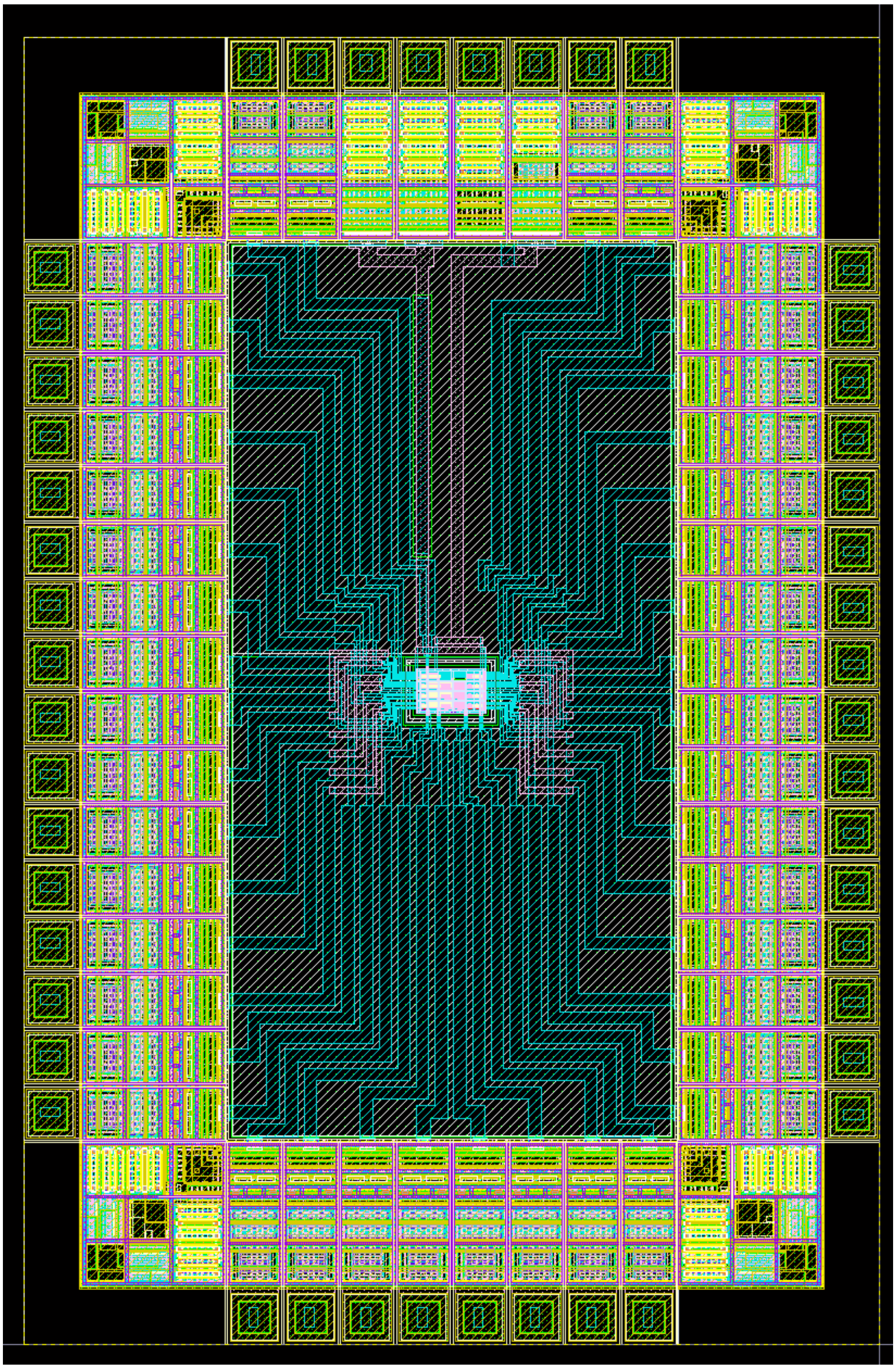

Figure 24: Layout of ALU with pad ring. The actual 4-bit ALU is contained in the small green both in the center of the figure.

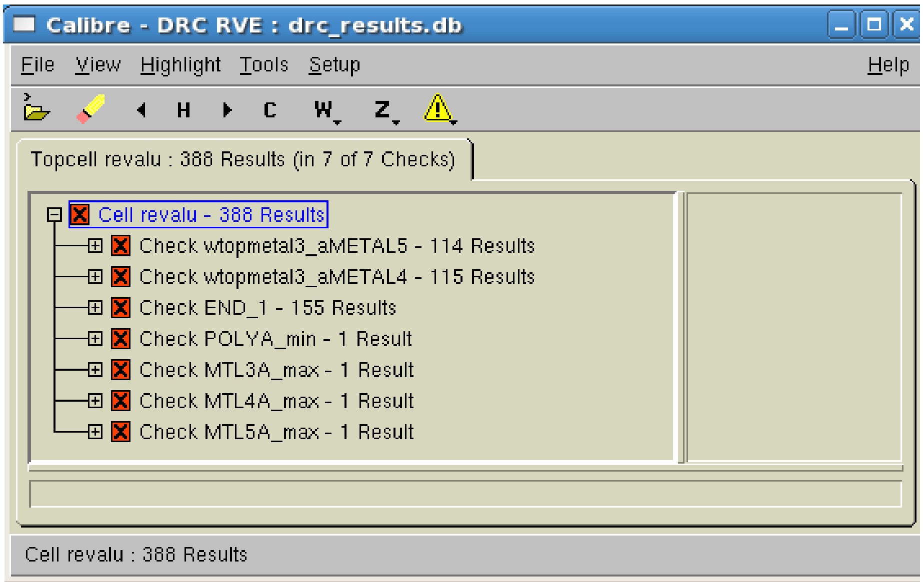

Figure 25: Design rule check for entire ALU chip design. The resulting seven different violations were expected and does not cause problems in the fabrication.

- **POLYA_min**: On all 1 mm^2 squares at least 5 % (and at most 60 %) must be covered by polysilicon. In our case we only use a small part of the chip and the rule is not violated if we only check the ALU circuit. We do, therefore, not have to do anything about it, but the normal solution is to add (unused) dummy polysilicon in the empty areas.
- **MTL3A_max** to **MTL5A_max**: After dummy metal has been added at most 60 % (and at least 20 %) of the area must be covered by metal at each layer. The errors here relate to metal layers 3 through 5, and as we use very little or none in the ALU layout it is easy to conclude that the error of too much metal comes from the naïve algorithm for placing dummy metal. After the layout have been send to manufacturing better algorithm will be applied that solves this and, thus, these errors does not cause problems.

5 Design of Reversible Circuits using Standard Cells and Functional Programming

The layouts of the reversible designs presented in this report and, to the authors knowledge, in all other literature have been implemented by hand. In Sec. 3.5 we started the process towards computer aided design by looking at standard cells of reversible gates. In this section we will look even further upwards in design chain and see how a recently proposed functional programming language can be used to aid implementating these circuits.

5.1 Current CAD Approaches to Reversible Logic

The first approach to computer aid in reversible logic designs were based on logic synthesis. It is not always easy to determine a good realization of a reversible circuit, for example, with respect to the number of garbage bits or transistor costs. In conventional logic, *logic synthesis* has been used for many years to find a good implementation for a given circuit definition. These methods can not be directly transferred to reversible logic, so redesigning these algorithms or even completely new synthesis algorithms for reversible circuits have attracted much attention [26, 19, 33, 20, 36]. Given a logic specification (*e.g.* as a logic table or a binary decision diagram), a reversible circuit is synthesized using a fixed library of basic reversible gates.

The first reversible design language is SyReC [37] and is based on the syntax of a reversible imperative language Janus [38]. A design is directly synthesized to logic through a number of steps that includes loop unrolling and translation of each statement and expression. This synthesis is, however, not always garbage free. Some translations (*e.g.* expression evaluation) will always generate garbage.

Both of these approaches target a flat netlist or diagram of reversible gates. This is not a problem for smaller circuits, but they lose the structure that was provided by the user. This information could be very useful in the placement of the standard cells.

5.2 A Combinator Description Language

A recent design language is a (point-free) combinator-style functional language and is designed to be close to the reversible logic gate-level [27]. The combinators, however, include high-level constructs such as ripples, conditionals, and, as it is a language to describe reversible circuits, a novel construct for inversion. The language is inspired by μFP [25], which is based on FP [3], but extended with memory with feedback loop. We will not describe the language in detail here (for those that are interested we refer to [27]), but only explain the combinators that are needed to design the ALU that was presented in the previous section.

The reversible gates are defined as atoms in the language and named Not, Feyn, Fred, and Toff. Subscripts on the gates denotes that the inputs are permuted and outputs are inverse permuted. The identity gate, Id, is also added as an atom. The basic ways to combine atoms is by a serial composition (written as $f\,;g$) or a parallel composition (written as $[f, g]$). We can then define an arbitrary-sized ALU as follows:

$$\begin{aligned}
rdn &= \mathsf{Feyn}_{\{3,6\}}\,;\mathsf{Feyn}_{\{5,6\}}\,;\mathsf{Fred}_{\{6,4,5\}} && (1)\\
rup &= \mathsf{Fred}_{\{6,4,5\}}\,;\mathsf{Toff}_{\{1,4,6\}}\,;\mathsf{Feyn}_{\{2,6\}} && (2)\\
group &= [\mathsf{Split}_4, \mathsf{Split}_2] && (3)\\
interface &= [\mathsf{Id}, \mathsf{Concat}\,;\mathsf{Zip}] && (4)\\
alu &= interface\,; && (5)\\
&\quad \diagdown(group\,; rdn\,; \{1,2,3,5,4,6\}\,; group^{-1})\,;\\
&\quad \diagup(group\,; \{1,2,3,5,4,6\}\,; rup\,; group^{-1})\,;\\
&\quad interface^{-1}
\end{aligned}$$

The first two equations, (1) and (2), are the two subpart that together make a bit-slice. These are the only logic gates in the definition. The next two equations, (3) and (4), are added to make the interface of the different construct match. There is, thus, no functionality in this, but can give suggestions to the routing. Split and Concat are grouping and ungrouping of wires, while Zip performs a merge of two arbitrary-sized input busses. The final equation, (5), implement the two ripples that is needed. Here, we use the inversion combinator with the grouping definition.

A feature of combinator languages is that they have a solid mathematical definition and algebraic laws that relate different combinators can be defined and proven. Using these relations in a clever way, it is possible to use term rewriting to optimize the circuit descriptions. A simple version of this technique have already been used for reversible circuit in what is called *template matching* [20]. Here the idea is to perform local optimizations by defining a large set of identity circuits and then match a subpart of the identity circuits with subparts of the circuit to be optimized. When the matched subpart of the circuit is larger than the rest of the identity circuit, the smaller subpart can be used instead without changing the functionality of the circuit. For the combinator language laws for the higher-level constructs (ripples *etc.*) also exist and, thus, it gives more possibilities for rewriting.

5.3 Combinators and Standard Cells

The combinator language is designed to be close to the logic gate level; atoms in the combinator language mirror the reversible standard cells. A translation to a netlist of reversible logic gates or other low-level descriptions would, therefore, be fairly straightforward and the translation would include flattening or unrolling, when specializing the circuit to a given input size. This approach will, however, suffer from the same problems as the previous reversible computer aided design approaches. We would end with a flat structure and then have to do *place and route* one each gate.

A better strategy is to keep and exploit the structure and information that already is in the combinator language. For example, the language contains combinators for downwards ($\diagdown g$) and upwards ripples ($\diagup f$) and these are the exact same structures that is used to implement the reversible ALU. In Sec. 4 we saw how a compact (and regular) implementation could be made by having each bit-slice in a separate row. By keeping the knowledge that we have a ripple, we can, therefore, easily make a good implementation of these circuits.

This can also be exploited when optimizing the circuit with term rewriting. Mostly we think of optimizations as reducing the number of gates or the circuit delay, but in this case it would also allow to improve the placement of cells (and to some extend routing) by finding more regular structures. When making the term rewriting system, we need to explicitly define priority and metric for the different algebraic laws. So if we can find a metric for improving the placement, it might be useful, but this is left as future work.

6 Conclusion

In this technical report, we have shown standard cell layouts for the basic set of reversible gates. The cells were designed to mirror the widely used diagram notation (left-to-right flow) for gates with up to three inputs. The cells were implemented in 0.35 μm CMOS using *complementary pass-transistor logic*. These cells are first prototype cells and knowledge for future improvements for CAD approaches have been gained from this work. At the heart of these improvements is to move the pins inside the cells.

As an example, the standard cells has been used to implement a (4-bit) reversible arithmetic logic unit. The circuit was fabricated, but before this, correctness of the layout were verified with simulations, *design rule check*, and *layout vs. schematic check*. After fabrication the resulting chip were tested for functional correctness.

The main purpose for using standard cells is to make computer aided designs much easier. We have here advocated to use a recent combinator-based reversible functional language and argued why this approach will be favorable over current approaches when it comes to circuits design. But much work is still needed in this area to know the benefits and drawbacks of the different approaches and even more work to have a complete working design flow.

It would also be desirable to have measurements of the fabricated chips that shows that the resulting chip did indeed use less energy than a conventional digital CMOS design, but this has not been the aim of this work. Better measurement equipment and a different chip design would be needed for this.

Acknowledgement

The author thanks Alexis De Vos and Stéphane Burignat for introducing this reversible logic family hand help with the chip fabrication. I also thank Robert Glück and Holger Bock Axelsen for discussions about this work. Finally, the author thanks the *Danish Council for Strategic Research* for the support of this work in the framework of the *MicroPower* research project. (`http://topps.diku.dk/micropower`)

References

[1] W. C. Athas and L. J. Svensson. Reversible logic issues in adiabatic CMOS. In *Workshop on Physics and Computation, PhysComp '94, Proceedings*, pages 111–118. IEEE, 1994.

[2] H. B. Axelsen and R. Glück. What do reversible programs compute? In M. Hofmann, editor, *Foundations of Software Science and Computational Structures*, volume 6604 of *LNCS*, pages 42–56. Springer-Verlag, 2011.

[3] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[4] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995.

[5] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[6] A. Bérut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.

[7] J. I. Cirac, L. Duan, and P. Zoller. Quantum optical implementation of quantum information processing. In F. De Martini and C. Monroe, editors, *Experimental Quantum Computation and Information*, Proceedings of the International School of Physics Enrico Fermi, pages 148–190. IOS Press, 2002.

[8] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton. A new quantum ripple-carry addition circuit. *arXiv:quant-ph/0410184v1*, 2005.

[9] R. Cuykendall and D. R. Andersen. Reversible optical computing circuits. *Optics Letter*, 12(7):542–544, 1987.

[10] A. De Vos. Reversible computing. *Progress in Quantum Electronics*, 23(1):1–49, 1999.

[11] A. De Vos and Y. Van Rentergem. Reversible computing: from mathematical group theory to electronical circuit experiment. In *Computing Frontiers Proceeding*, pages 35–44. ACM Press, 2005.

[12] R. P. Feynman. Quantum mechanical computers. *Optics News*, 11:11–20, 1985.

[13] M. P. Frank. Common mistakes in adiabatic logic design and how to avoid them. In H. Arabnia and L. Yang, editors, *Proceedings of the International Conference on Embedded Systems and Applications. ESA'03*, pages 216–222. CSREA Press, 2003.

[14] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3-4):219–253, 1982.

[15] J. Koller and W. Athas. Adiabatic switching, low energy computing, and the physics of storing and erasing information. In *Workshop on Physics and Computation, PhysComp '92, Proceedings*, pages 267–270, 1992.

[16] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[17] A. Lines. Pipelined asynchronous circuits. Technical Report CaltechCSTR:1998.cs-tr-95-21, California Institute of Technology, 1998.

[18] A. Lines. Asynchronous interconnect for synchronous SoC design. *IEEE Micro*, 24(1):32–41, 2004.

[19] D. Maslov and G. W. Dueck. Reversible cascades with minimal garbage. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(11):1497–1509, 2004.

[20] D. Maslov, G. W. Dueck, and D. M. Miller. Synthesis of Fredkin-Toffoli reversible networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(6):765–769, 2005.

[21] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, second edition, 1980.

[22] R. C. Merkle. Reversible electronic logic using switches. *Nanotechnology*, 4(1):21–40, 1993.

[23] M. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[24] P. Patra and D. S. Fussell. A framework for conservative and delay-insensitive computing. Technical report, 1996.

[25] M. Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 104–112. ACM, 1984.

[26] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes. Synthesis of reversible logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):710–722, 2003.

[27] M. K. Thomsen. Describing and optimizing reversible logic using a functional language. In A. Gill and J. Hage, editors, *Implementation and Application of Functional Languages, 23rd International Workshop, IFL 2012*. LNCS, 2012. To appear.

[28] M. K. Thomsen and H. B. Axelsen. Parallelization of reversible ripple-carry adders. *Parallel Processing Letters*, 19(1):205–222, 2009.

[29] M. K. Thomsen, H. B. Axelsen, and R. Glück. A reversible processor architecture and its reversible logic design. In A. De Vos and R. Wille, editors, *Reversible Computation, RC 2011. Revised Selected Papers*, volume 7165 of *LNCS*, pages 30–42. Springer-Verlag, 2012.

[30] M. K. Thomsen, R. Glück, and H. B. Axelsen. Reversible arithmetic logic unit for quantum arithmetic. *Journal of Physics A: Mathematical and Theoretical*, 43(38):382002, 2010.

[31] T. Toffoli. Reversible computing. In J. W. de Bakker and J. van Leeuwen, editors, *ICALP*, volume 85 of *LNCS*, pages 632–644. Springer-Verlag, 1980.

[32] Y. Van Rentergem and A. De Vos. Optimal design of a reversible full adder. *International Journal of Unconventional Computing*, 1(4):339–355, 2005.

[33] Y. Van Rentergem and A. De Vos. Synthesis and optimization of reversible circuits. In Reed-Muller, editor, *Proceedings of the Reed-Muller Workshop 2007*, pages 67–75, 2007.

[34] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147–153, 1996.

[35] N. H. E. Weste and D. Harris. *CMOS VLSI Design : a circuit and system perspective*. Addison-Wesley, Pearson Edication, third edition, 2005.

[36] R. Wille and R. Drechsler. *Towards a Design Flow for Reversible Logic*. Springer Science, 2010.

[37] R. Wille, S. Offermann, and R. Drechsler. SyReC: A programming language for synthesis of reversible circuits. In *Specification & Design Languages, FDL 2010. Forum on*, pages 1–6. IET, 2010.

[38] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Program Manipulation. Proceedings*, pages 144–153. ACM Press, 2007.