The Compilation of Reversible Circuits and a New Optimization Game

by

Alex Parent

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Physics

Waterloo, Ontario, Canada, 2016

Abstract

The focus of this thesis is reversible circuit compilation.

We will explore the use of pebble games for circuit analysis. The usefulness of this technique is demonstrated by finding a new space bound for the Karatsuba algorithm and more generally for any similar algorithm based on recurrence relations. A new pebble game based on the reversible pebble game which better captures the use of in-place operations is also presented. We also construct circuit to compute trigonometric functions based on the CORDIC algorithm and analyze it using this game.

Acknowledgements

I would like to thank my supervisor Michele Mosca for his guidance, insight, and for introducing me to the field of Quantum Information. I would also like to thank members of my defence and advisory committees Agata Branczyk, Richard Cleve, and Matteo Mariantoni for providing helpful advice and suggestions. Thanks to George Labahn for suggestions regarding algorithms for the computation of trigonometric functions.

I also wish to thank Martin Roeteller who was a primary collaborator on much of the work contained in this thesis. The experience and guidance I gained working with him during internships at both NEC and Microsoft was invaluable.

Thanks to my office mate Vincent Russo, Matt Amy, and members of the QSoft group for helpful discussions at various points during my research.

I would also like to thank my parents, Brian and Mary-Lynn Parent, for supporting and encouraging my education.

Finally I would like to thank Natalia Smiarowski for her love, support, and help proofreading this thesis.

Table of Contents

List of Figures

Chapter 1

Introduction

Reversible computing initially gained interest as a way of avoiding the fundamental energy requirement imposed by *Landauer's Principle*. It was shown, initially in 1963 by Yves Lecerf [15], then again independently by Charles Bennett [3] in 1973, that any classical irreversible computation could be made reversible[1]. Interest in reversible computing has grown recently due to its applications to quantum computing.

In this thesis the focus will be on the compilation of reversible circuits in the context of quantum computing. Tools for efficient circuit compilation are necessary in the long term for implementing high level algorithms on a quantum computer as well as in the short term for resource estimation[2].

Compiling an algorithm for a quantum computer is a multi-layered process. Efficient implementations of operations are needed at the physical, fault-tolerant and logical levels. An important part of compilation at the logical level is the generation of reversible circuits. The goal of this thesis is to describe a framework for the efficient compilation of high-level programs into low level reversible circuits. It will discuss the use of pebble games for circuit optimization. Also introduced is a new method for analysis in reversible computation using a modified version of the reversible pebble game.

[1]It is shown in both papers that a reversible Turing machine can simulate an irreversible with some space overhead.

[2] Resource estimation is very important for example in Quantum Safe Cryptography. Parameters for crypto algorithms must be chosen to resist generic quantum search attacks. Without access to cost estimates implementers are forced to choose overly conservative parameters which can lead to extra computation time and/or bandwidth use in cryptosystems.

1.1 Landauer's Principle

Landauer's principle [14] states that any logically irreversible operation must be accompanied by a corresponding entropy increase in non-information bearing degrees of freedom. Another way to state this is that any operation which does not preserve information must have an associated energy cost. This can be understood through the second law of thermodynamics which states that the entropy of a closed system cannot decrease. That is to say the number of possible states a closed system might be in cannot decrease.

The minimum bound on the amount of energy required to perform an irreversible operation is called the Landauer limit ($kT \ln 2$). Using reversible computing it is in principle possible to avoid this cost by ensuring that no information is lost in the computation.

1.2 Reversible Computation

Normally when performing computations information is discarded. Actions such as overwriting and clearing memory cause many states to be mapped to the same state. So even if one had knowledge of the output operations involved in a computation it may not be possible to recover the initial state. These types of computations are therefore called irreversible.

A computation is called reversible if any state can be traced uniquely forwards and backwards through time, or equivalently if each step in the computation is bijective. Bennett [3] showed that in principle any computation can be made reversible.

Reversible computing is relevant in the context of quantum computation since it is necessary to be able to implement unitary operations[3]. Efficient generation of quantum circuits is important since in many quantum algorithms the cost of implementing a reversible oracle dominates. For example when Grover's algorithm is used to query a classical function the query must be implemented as a reversible circuit. Since the diffusion operator is relatively small compared to the size of many functions of interest [8, 1] achieving a small circuit size is very important.

The focus of this thesis is on efficiently generating the reversible parts of a quantum computation using the circuit model. In a quantum computer conditional operations require measurements to be made. The circuit model corresponds with measurement free computation since all operations are applied unconditionally.

[3] In the classical domain unitary operations are equivalent to reversible operations.

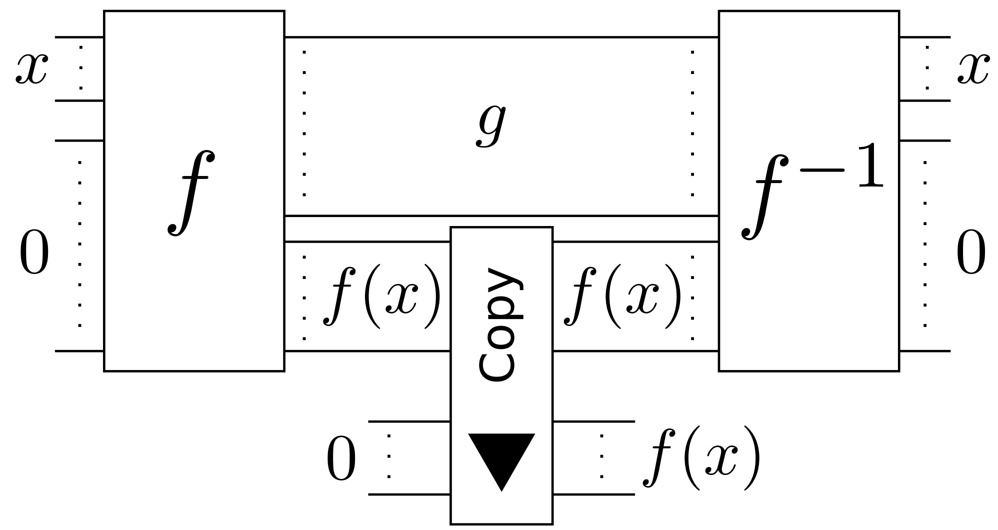


Figure 1.1: Bennett Method. Performs the operation $(x, 0^n, 0^m) \mapsto (x, f(x), 0^m)$ where n is the size of the output and m is the number of ancilla needed for the computation of $f(x)$ Copy uses a cascade of CNot gates, i.e. it maps basis states: $(x_1, \ldots, x_n, 0^n) \mapsto (x_1, \ldots, x_n, x_1, \ldots, x_n)$.

In order for an operation to be considered reversible it must be forwards and backward deterministic[4]. For example Consider a classical computation where we have inputs a, and b and wish to know the output $a \wedge b$. The classical AND gate computes $(a, b) \mapsto (a \wedge b)$, discarding the values of a and b. Since this function is not bijective (ex. $0 \wedge 0 = 0$ and $1 \wedge 0 = 0$) it is not reversible. Similarly the fanout gate $a \mapsto (a, a)$ is also not considered reversible. This is because it is not backwards deterministic on all inputs; for example given an input $(0, 1)$ the reverse fanout gate has no defined output.

One way to make a reversible AND operation is to instead use the bijective function $(a, b, c) \mapsto (a, b, a \wedge b \oplus c)$. This is called the "Toffoli Gate". (If c is initialized to zero we have $(a, b, 0) \mapsto (a, b, a \wedge b)$).

Any irreversible function can be made reversible using the "Bennett Method" [3]. In this method each irreversible gate is replaced with a reversible version with additional space allocated as necessary. Then the final result is copied on to the newly allocated set of bits and the entire circuit is repeated in reverse in order to uncompute any intermediate information computed by the circuit(see fig. 1.1). So given an irreversible function $f(x)$ (where $f(x)$ is of size n) we can use the Bennett method to generate a reversible function $(x, 0^{n+m}) \mapsto (x, f(x), 0^m)$.

[4]In other words it must be bijective

The standard reversible gate set which will be used includes the following gates:

$$\begin{aligned}
&\text{Not:} a \mapsto \neg a \\
&\text{Cnot:} (a, b) \mapsto (a, a \oplus b) \\
&\text{Toffoli:} (a, b, c) \mapsto (a, b, ab \oplus c) \\
&\text{Swap:} (a, b) \mapsto (b, a) \\
&\text{Fredkin:} (a, b, c) \mapsto (a, \neg ab \oplus ac, ab \oplus \neg ac)
\end{aligned}$$

Note another way of describing the Fredkin gate is as swap on b and c controlled on the value of a:

$$\begin{aligned}
(0, b, c) &\mapsto (0, b, c) \\
(1, b, c) &\mapsto (1, c, b)
\end{aligned}$$

1.3 Quantum Circuits

Analysis will be done using the Clifford plus T gate set. This choice is made due its availability in error correction schemes such as the surface code [7]. Note that this gate set does not include the Toffoli gate so it will have to be synthesised from more primitive gates in the set. As shown in [2] this can be done using seven T-gates in T-depth 4.

Long range interactions are assumed to be inexpensive in this model compared to the cost of T gates.

Chapter 2

Memory Management and Pebble Games

Memory management is important when compiling a program to be run in a resource constrained environment. It is a topic of great interest in program compilation in general. The problem is however simpler in the case of irreversible languages. Once it is determined that some data is no longer needed it can be zeroed and reallocated. In a reversible computation memory management is somewhat complicated by the reversibility constraint since a bit cannot be simply set to zero and reused[1].

Pebble games provide a convenient abstraction for modelling both reversible and irreversible memory models. A pebble game is played by placing pebbles onto the nodes of a graph. The goal of such a game is generally to place a pebble on a specific node or set of nodes. In the reversible pebble game the rules governing pebble placement are chosen such that the game models memory in reversible computation. Once a particular pebbling strategy is found it can be used to generate a circuit for the computation represented by the graph as shown in fig. 2.1(this example will be further explained later in this section). The difference between the reversible and irreversible memory models is perhaps made most clear by the rules of the black and reversible pebble games discussed in section 2.1.

A new game for modelling reversible computation played on a *Mutable Dependency Diagram* (MDD) is presented. The MDD representation contains all necessary information about the structure of the computation that is required for the compiler to perform automated space-time optimizations using pebble games. It is proposed as an intermediate representation to be used when compiling high level programming languages to reversible

[1]This is because the operation $x \mapsto 0$ is not reversible.

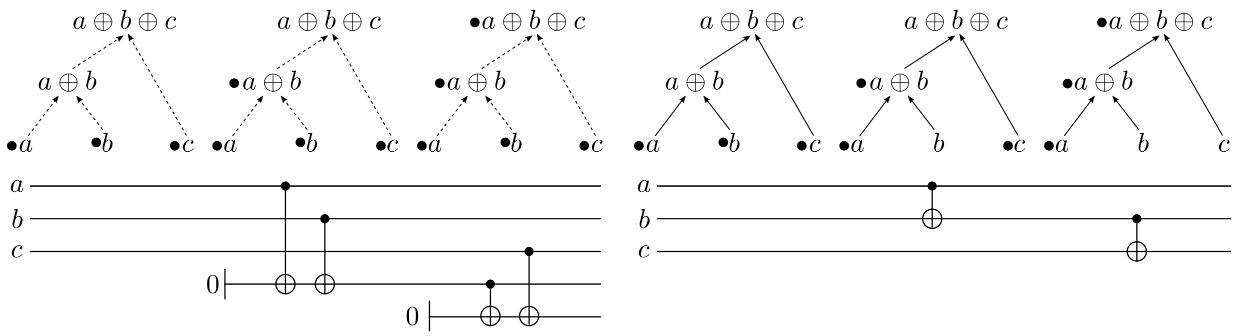


Figure 2.1: A reversible pebble game (left) and MDD (right) for computing $a \oplus b \oplus c$. Mutation edges are represented with solid lines while normal dependency edges use a dashed line. Adding a new pebble corresponds to adding an ancilla qubit initialized to 0. Below each move the gates associated with that move are shown. Note instead of moving the c pebble to $a \oplus b \oplus c$ the $a \oplus b$ pebble could have been moved. This would result in a slightly different circuit (with both CNot gates targeted on b) that would still produce the output $a \oplus b \oplus c$.

circuits. This representation is shown to be useful not only in the context of compilation but also in the analysis of circuits.

2.1 Pebble Games

Any computation where execution is independent of input can be modeled as a Directed Acyclic Graph (DAG). This graph is sometimes called a *Functional-dependence* network. The nodes on the graph are associated with functions and the arrows represent data dependencies required for the computation of those functions. Such a graph is loop free by construction if there are no circular dependencies[2]. Pebble games can be used in conjunction with these graphs to explore time-space trade-offs in circuit generation.

Each pebble represents some amount of space[3]. Placing a pebble corresponds to computing the value of the node and storing it in the space represented by the pebble. Removing a pebble corresponds to clearing the value which was stored in its space and making it available to be used again. In order to generate efficient circuits it is important to minimize both the number of moves and the maximum number of pebbles in play.

[2]Any computation with circular function dependencies is not well-formed and is not implementable.

[3]The amount of space need not be identical for each pebble. A pebble takes up the amount of space that is required to store the value that it represents.

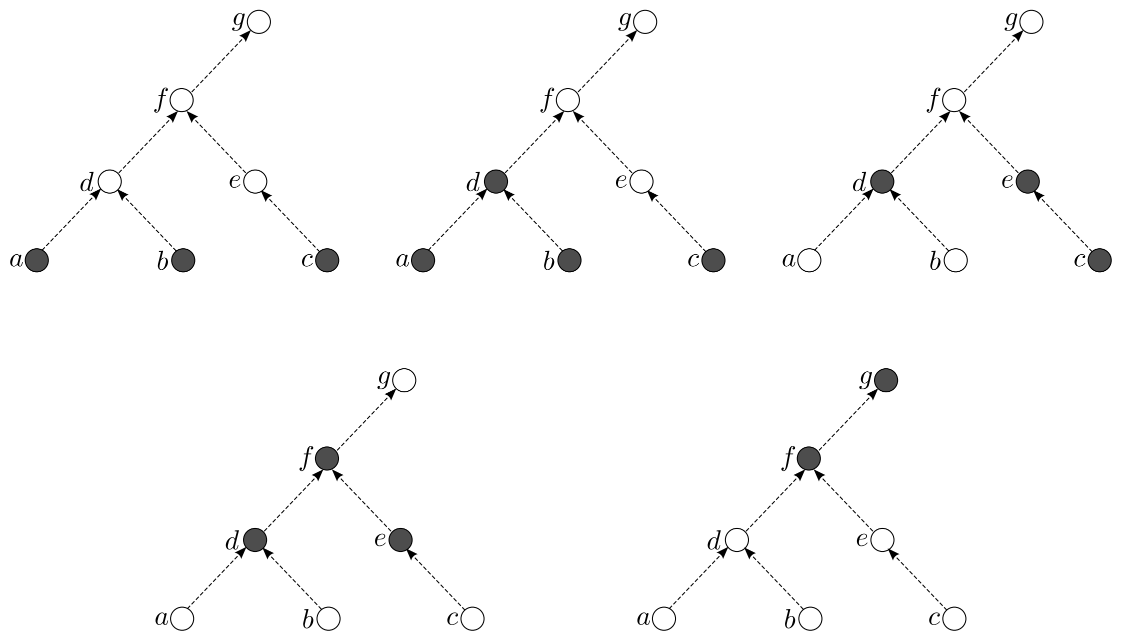


Figure 2.2: Black Pebble game (used in irreversible computation). This figure demonstrates a strategy for pebbling the g node using a total of four pebbles.

2.1.1 Black Pebble game

The *Black Pebble Game* is used to model irreversible computation. In irreversible computing we must have a functions data dependencies to compute its result but we can clear and reuse space at any time. Therefore the rules are as follows [9],(an example is shown in fig. 2.2):

1. A pebble can be added to a node if and only if all predecessors of the node have pebbles.
2. A pebble can be removed at any time.

2.1.2 Reversible Pebble Game

Bennett [4] describes an alternative pebble game for reversible computation called the *reversible pebble game*. The rules are similar to those used in the *black pebble game*

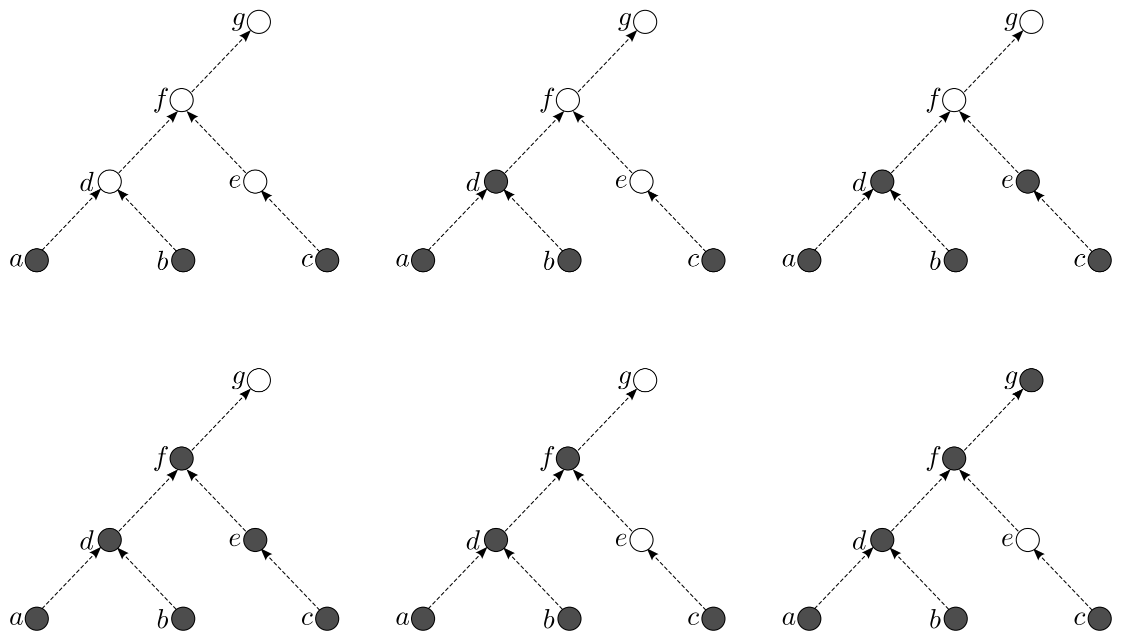


Figure 2.3: Reversible Pebble game. This figure demonstrates a strategy for pebbling the g node using a total of six pebbles.

except that the reversibility constraint prevents us from simply removing pebbles. Each computation has a corresponding reverse computation, and the dependencies of the reverse computation are the same as the corresponding forward computation. This means that pebbles may still be removed, but removal is subject to the same conditions as placement. In other words we make placement and removal of pebbles symmetric similarly to the way forward and reverse computation are made symmetric in reversible computing. The rules for this version of the game are given below (an example is shown in fig. 2.3):

1. A pebble can be added to a node if and only if all predecessors of the node have pebbles
2. A pebble can be removed from a node if and only if all predecessors of the node have pebbles

The space complexity of a game is given by the maximum number of pebbles which are in play at any given time. It has been shown that in general computing the *pebble* *number* (the minimum number of pebbles, over all possible games, required to the final target pebble) for an arbitrary DAG is PSPACE-complete[4][5].

Strategies for playing these games have been analyzed in the case of, one dimensional directed graphs, as well as for trees [13]. There are three particular strategies of interest in the simple 1D case where all nodes have equal cost (fig. 2.4 represents these strategies visually):

The first is the naive strategy (sometimes called *Bennett Method*). In this strategy pebbles are added one after another until the desired pebble is reached. All pebbles besides the one on the desired node are then removed in reverse order. This strategy has time complexity $2n - 1$ and space complexity n. This is the minimal time strategy in the 1D graph. The equivalent strategy on a general DAG might be to fill in all pebbles in topological order then remove all non-output pebbles in reverse order.

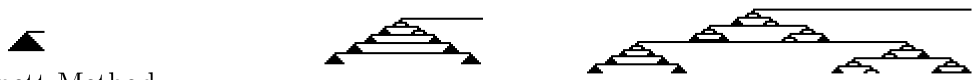
The second is a slightly more advanced strategy which will be referred to as the *incremental strategy*. Here we start with some limited number of pebbles and place pebbles until we run out. After running out we uncompute all but the last pebble following the Bennett strategy. This is then repeated using the next pebble as the new starting point until the desired pebble is reached. This strategy has a time complexity of $O(n)$ and a space complexity of $O(\sqrt{n})$. This means that we can can get a square root reduction in space in exchange for a constant multiplier on time, therefore the asymptotic space-time product of the circuit can be improved over the naive strategy. Although this strategy is not optimal in all cases it is easy to implement due to not requiring that the cleanup algorithm have prior knowledge about the size of the circuit. A similar strategy applied to trees is used in section 3.2.2 in the analysis of reversible Karatsuba.

Finally the third is the optimal strategy (for a given space bound) from Knill [12]. Finding such a strategy in the general case is PSPACE-complete.

2.2 Mutable Dependency Diagram (MDD) [18]

In this section a new game for the optimization of reversible computation is presented. It is a generalization of the reversible pebble game in that any reversible pebble game can be represented on an MDD.

[4]A problem is said to be in PSPACE if it can be solved using an amount of memory which is polynomial in the input length. A problem is PSPACE-complete if all problems in PSPACE can be transformed into it using a polynomial amount of time. The PSPACE-complete class contains the well known Quantum Merlin Arthur (QMA), BQP, and NP-complete classes and is equivalent to Quantum Interactive Polynomial time (QIP).



(a) Bennett Method **(b)** Incremental **(c)** Knill

Figure 2.4: Comparison of 1D pebble strategies. The horizontal axis represents time and the vertical axis is the state of the game at a given time slice.

Lacking in the original pebble game is an effective way to represent mutation[5]. This is of not much consequence in the irreversible pebble game since mutation can be modeled as a placement followed by the removal of a dependency. In the reversible pebble game it is significantly more important since pebbles can normally only be removed if predecessors are pebbled. For example addition via the mapping $(a, b) \mapsto (a, a + b)$ (implemented by an in-place addition circuit) is a reversible operation which does not require new space to be allocated for the result $a + b$. In order for strategies from the reversible pebble game to be applied to a circuit using this operation generically it would have to be made out of place by copying the b input and thus losing the advantages that may have been gained by using an in-place operation.

In order to better represent in-place operations a new pebble game is proposed. Rules one and two are similar to the reversible pebble game. Two types of labeled edges and a third rule have been added (see fig. 2.1 for an example of an MDD game):

1. A new pebble can be added to a node if and only if all predecessors of the node have pebbles.
2. A pebble can be removed entirely if and only if all predecessors of the node have pebbles.
3. A pebble may be moved forward or backward along a mutation edge if all predecessors of the node to which the edge points have pebbles.

Note that since moving a pebble as described in the third rule is optional this is a generalization of the reversible pebble game and any strategy that can be used in the reversible pebble game is a valid strategy here. We also inherit the PSPACE-completeness ([5]) of solving the reversible pebble game since any graph which does not include mutation edges is played by the rules of the reversible pebble game. This means that in general solving an MDD must be at least as hard.

[5]Mutation in this context means variables that change in value. For example the modification operators operators discussed in section 2.2.1

As a simple example of the usefulness of such a game consider the computation of $a \oplus b \oplus c$. In fig. 2.1 the reversible pebble graph as well as the MDD are given. If the inputs (a,b,c) are considered to be pebbled then two additional stones are needed to pebble $a \oplus b \oplus c$ in the reversible pebble game. The first pebble is placed on $a \oplus b$ and the second can then be placed on $a \oplus b \oplus c$. The MDD game on the other hand does not need any additional stone. In the MDD the b stone can be slid along the modification path to $a \oplus b$, then the c stone can be moved to $a \oplus b \oplus c$. So it is possible to use two fewer stones (no additional stones beyond the inputs) in the MDD game. Note that since this game is a generalization of the reversible pebble game the best strategy for an MDD will always be equivalent to or better then the best strategy in the corresponding reversible pebble game.

Theorem 1. *For all MDDs a strategy to pebble all nodes exists and is efficiently computable.*

Proof. Consider the following simple strategy. First perform a topological sort[6]on the nodes of the graph. Now place a pebble on each node in the sorted order. It is guaranteed that the dependencies of each node will be pebbled since they are placed in sorted order. The entire graph may be covered by pebbles in this fashion. □

2.2.1 Converting a Computation to an MDD

First I will define the elements of the graph in terms of what they represent with respect to computation:

Nodes represent a register in some state.

Mutation Edges represent operations which transform the value of a register. The parent and child node of an edge represent the state of the register before and after (respectively) the computation. Each node may have only a single input and output mutation edge. A chain of nodes connected by mutation edges represents all computations which change the contents of a register.

[6]A topological ordering is an ordering of nodes in a directed graph such that for all edges ab; a comes before b. A DAG always has at least one topological ordering. The complexity of finding a topological (i.e. performing a topological sort) ordering is $O(E+V)$ where E is the number of edges and V is the number of vertices in the graph.

Dependency Edges point from values upon which a computation depends. In order for a computation defined by a mutation edge to be implemented all parent nodes along dependency edges pointing to the child node of the mutation edge must be available.

Example: Janus

Janus [23, 16] is a reversible programming language. It's main focus is not to be a circuit description language, but to be logically reversible. As a result of logical reversibility it does however have some useful properties which help when compiling to circuits.

Janus guarantees that all programs implemented in it are reversible. This reversibility is implemented at the statement level, that is to say that all statements are reversible but the internals of a statement may not be. This means that temporary ancilla may be allocated to compute a statement but this ancilla can be cleaned up between statements.

Modification operators are used for operations that can be done in-place[7]. For example the addition modification operator, `+=`. Operations that cannot be done in-place can only occur to the right of a modification operator. These operations can then be implemented, the result can be applied in place to the left hand side value, and then cleaned up using the Bennett method if necessary. This means that memory need only be allocated temporarily for each modification statement.

To illustrate with and example, Say we wish to evaluate the statement `b+=a*c`[8]. First `a*c` is computed by using additional ancilla as needed. The result is then reversibly added to `b` (using the function $(a, b) \mapsto (a, a + b)$ discussed previously). Finally the circuit for `a*c` can be reversed in order to clean up any allocated ancilla.

For example consider the Janus program (where `x`, `y`, and `z` are the inputs):

```
x y z
t = x + 5
y += t
z += y * t
x += z + y
```

[7]An in-place operation is one which overwrites one of its inputs with the result. For example the previously mentioned adder which maps $(a, b) \mapsto (a, a + b)$ is in-place. Conversely an out-of-place operation computes its result onto a set of ancilla and leaves the inputs unchanged. For example an out-of-place adder could compute the function $(a, b, 0^n) \mapsto (a, b, a + b)$.

[8]i.e. replace the bit-string b with the value $b + ac \mod 2^n$ where n is the length of b.

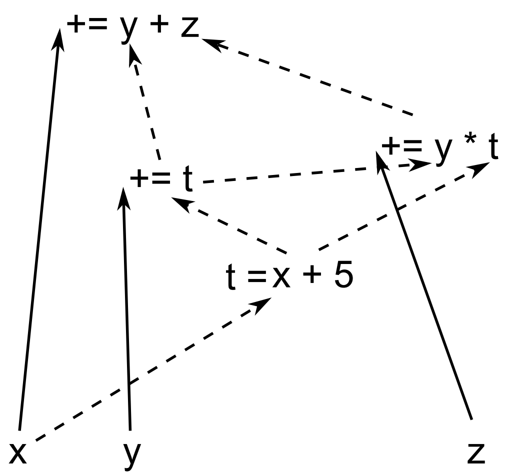


Figure 2.5: Example of an MDD generated from a Janus program.

To construct an MDD from this program the following procedure is repeated for each statement in the program:

1. Take the modification operator and the statement on the right add a corresponding node to the MDD. For example for the statement `a += b + c` add the node `+= b + c`.
2. Draw a modification arrow from the current value of the variable on the left of the operator to the node for that statement. In our example modification arrow is drawn from the node currently labeled `a` to the node `+= b + c`.
3. Draw a dependency arrow for each of the variables nodes statement from the current value of that variable to the node. In our example dependency arrows are drawn from the nodes labeled `b` and `c` to the node `+= b + c`.
4. Set the current label of the variable on the left of the operator to the node. So the label for `a` is moved to the node `+= b + c`.

For our example this will result in the MDD shown in fig. 2.5.

2.2.2 Computing the Cost of a Strategy

The goal of a pebble game is to come up with a set of moves the minimizes the total movement cost as well as the space complexity. So it is required that we define some method for computing these costs. In order to do this a few additional rules are needed:

4. Each node has a computation cost which is paid when placing a new pebble on it or sliding an existing pebble onto it.
5. Each node has a space cost which gives the contribution of that node to the overall space complexity when it is pebbled.
6. If a node has no parent it computes a constant so a pebble may be placed at any time.

A implementation of a computation can be represented by an MDD and a list of moves $\{m_i : i \in 1 \ldots n\}$ (where n is the total number of moves making up the computation). Each move consists of a graph location and a move type. Given a function C which returns the gate cost of a move the total gate cost of a computation is then:

$$\sum_{i \in 1 \ldots n} C(m_i)$$

Another way to represent the computation is as a set of states representing the graph after each move $\{g_i : i \in 1 \ldots n\}$. Given a function S which returns the total space in use for a state (by summing over the cost of all currently placed pebbles) we can compute the space complexly as:

$$\max_{i \in 1 \ldots n} S(g_i)$$

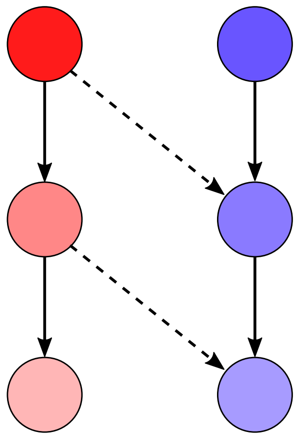
When implementing the graph as a circuit we may perform the computations in any valid topological ordering. Further if there are multiple computations which could be next in a given ordering they can be performed in parallel.

Some heuristic strategies are discussed below.

2.2.3 Eager Cleanup

Eager Cleanup is an example of a heuristic strategy for MDD pebbling.

If a value is no longer needed in a computation it can then be checked that the information needed to clean it up is *available.* Available is defined to mean that all dependencies needed


(a) One Way

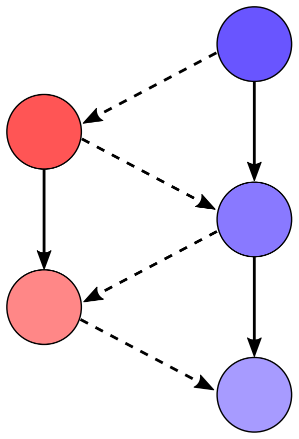

(b) Interdependent

Figure 2.6: Dependency patterns which affect eager cleanup.

to move the pebble to the beginning of its *modification path*[9]currently have pebbles or have pebbles that can be moved into place (by sliding along mutation edges) without causing additional space to be used. If this is the case it can immediately be cleaned and the ancilla can be freed for future use in the computation.

One way to implement this strategy is to first pebble the graph using some other strategy then to look back on the set of moves and see when each pebble was no longer required as a dependency in future moves. Then check if it includes only one way dependencies. If so check the time cost of zeroing the bit and do so if it is below some threshold.

Eager cleanup is possible when all dependencies are *one-way*. This means that once there is no path from it to any of the modification paths of its dependencies (as shown in fig. 2.6a).

Theorem 2. *Eager cleanup is possible on any graph with only one-way dependencies. Let $G = (V, E)$ be an MDD. Assume that all mutation paths in G are one-way dependent. The* Eager *cleanup method correct in the sense that any pebble with only one way dependencies can be removed at any time for some cost in time without incurring any additional space cost in the process.*

Proof. Consider a graph which consists of one-way dependent paths $P_1, \ldots, P_n$ arranged in topological order. Assume that for a P_k we can move the pebble in each path to an arbitrary position on the path.

[9]Modification path refers to a set of nodes chained together with solid modification arrows.

For the base $k = 1$ there is only one mutation path, this path either leads to an output, in which case no cleanup is necessary, or leads to a node that has to be cleaned up. However, since all inputs to the path are still available by assumption, the pebble on this path may be moved up and down and may therefore be moved to an arbitrary position.

Now, we make the inductive step to $k + 1$ paths. Recall that a pebble can be moved backward along a path if all their dependency edges pointing to the node that it occupies point from nodes which contain pebbles. We therefore make the inductive step as follows: by assumption on the one-wayness of the graph, all edges point into P_{k+1} and none points backward. Now, consider all nodes in $P_1, \ldots, P_k$ that would have to be pebbled in order to move the pebble on path P_{k+1} one step backward or forward. By induction, starting with P_k, we can slide the pebbles on each path into the location that is needed to move the pebble on P_k. By repeating this we may move the pebble on P_{k+1} to any location on the graph. $\square$

2.2.4 Triangles and Incremental Cleanup

Figure 2.6b shows a graph where eager cleanup fails. The red mutation path depends on previous values of the blue path in order to be cleaned.

It is still possible to clean up the blue node. A method for doing this is somewhat related to the *incremental strategy* discussed above generalized to a DAG.

In this strategy we start adding pebbles in some topological order working toward the final pebble. After we have placed a certain number of pebbles (up to some space limit we are trying to achieve), we stop and find the set of the currently pebbled nodes that are dependencies of future nodes. We then create a 'checkpoint' removing all nodes except the ones in this set by reversing the computation. Using the newly freed up pebbles the computation can be continued. If we run out of pebbles again we can create a new checkpoint and remove all pebbles up to the previous one and so on. Note that it is helpful to choose checkpoint locations where the set of nodes with future dependencies is small.

This strategy introduces a constant multiplier on time in exchange for a quadratic reduction in space in some cases (for example in the 1D case). This is not necessarily the case for all graphs though. Later discussion in section 3.2.2 shows a smaller then quadratic space reduction when applying this strategy to the Karatsuba circuit.

2.2.5 Circuit Generation

When using an MDD to construct a circuit additional information must be added to each node specifying which circuit input corresponds to each incoming arrow.

We start with a circuit consisting of registers corresponding to the input nodes of the graph, these are the inputs to the circuit. Ancilla are allocated using a heap[10], this ensures that the lowest available numbered ancilla is always the one allocated so ancilla use is minimized.

Starting with a set of built in functions, new functions can be constructed as MDDs. After a function is constructed it can be used a graph node in an MDD.

- Place Pebble: Place circuit which computes the function at the node using the nodes dependencies as input. Assign ancilla for the output of the circuit from the heap.
- Remove Pebble: Place the reverse circuit for the function at the node using dependencies as input, uncomputing the previously computed value. Return previously assigned ancilla to the heap (ancilla assigned when the pebble was placed).
- Slide Pebble: Place circuit which computes the function at the node using the nodes dependencies as immutable input and the bits representing the current value of the node as the modified input.

2.2.6 Example: SHA-256 Round

A simple statement of the computation done in one round of SHA-256 is given in algorithm 1. A direct translation of this algorithm into an MDD is given in fig. 2.7. One immediate advantage of the representation is that computation on the h and d registers may be modified instead of being reassigned at each step. It can also be seen from the MDD that the computed temporary values (Maj, Ch, Σ_0,Σ_1) all have one way dependencies and depend only on unmodified input values. The can therefore be cleaned up directly after they are used using the eager cleanup scheme. The resulting circuit is shown in fig. 2.8.

Note that the order of operations chosen for the circuit could have different based on the MDD. The compiler is allowed to exploit any ambiguity allowed by the representation to implement more optimized circuits.

[10]A heap is a tree data structure where a parent node always has a lower (for a min heap) or higher (for a max heap) value then its children. The root node of a min heap is the lowest valued node in the heap. This data structure is used here to always efficiently assign the lowest valued ancilla.

Algorithm 1 SHA-256

```
for $i \leftarrow 0, 63$ do
    $\Sigma_1 = (\mathbf{E} \ggg 6) \oplus (\mathbf{E} \ggg 11) \oplus (\mathbf{E} \ggg 25)$
    $\mathbf{Ch} = (\mathbf{E} \wedge \mathbf{F}) \oplus (\neg\mathbf{E} \wedge \mathbf{G})$
    $\Sigma_0 = (\mathbf{A} \ggg 2) \oplus (\mathbf{A} \ggg 13) \oplus (\mathbf{A} \ggg 22)$
    $\mathrm{Maj} = (\mathbf{A} \wedge \mathbf{B}) \oplus (\mathbf{A} \wedge \mathbf{C}) \oplus (\mathbf{B} \wedge \mathbf{C})$
    $\mathbf{H} \mathrel{+}= \Sigma_1 + \mathbf{Ch} + \mathbf{K}[i] + \mathbf{W}[i]$
    $\mathbf{D} \mathrel{+}= \mathbf{H}$
    $\mathbf{H} \mathrel{+}= \Sigma_0 + \mathrm{Maj}$
end for
```

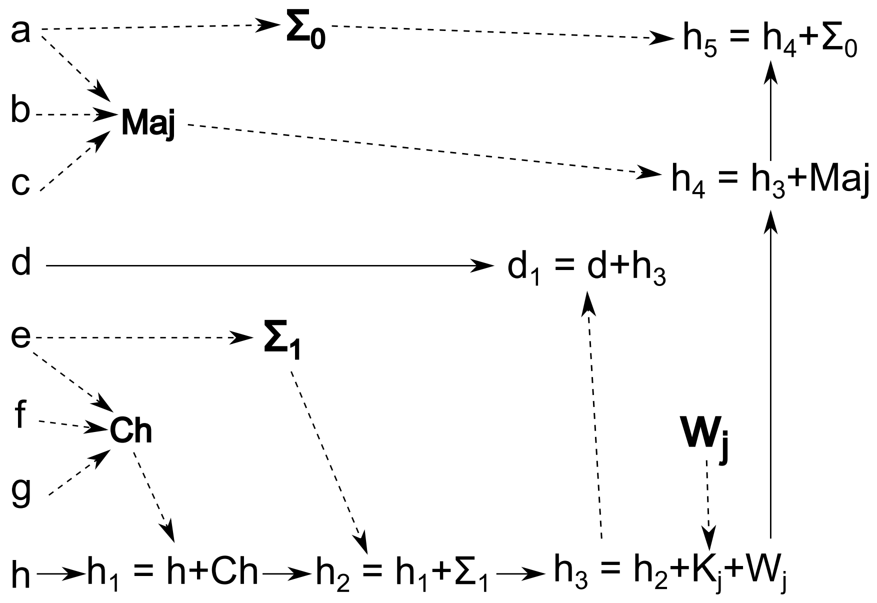


Figure 2.7: MDD for one round of SHA-256 generated using the eager cleanup method. Note for example that a pebble can be placed on **Ch** and then removed after $\mathbf{h}_1 = \mathbf{h} + \mathbf{ch}$ is computed.

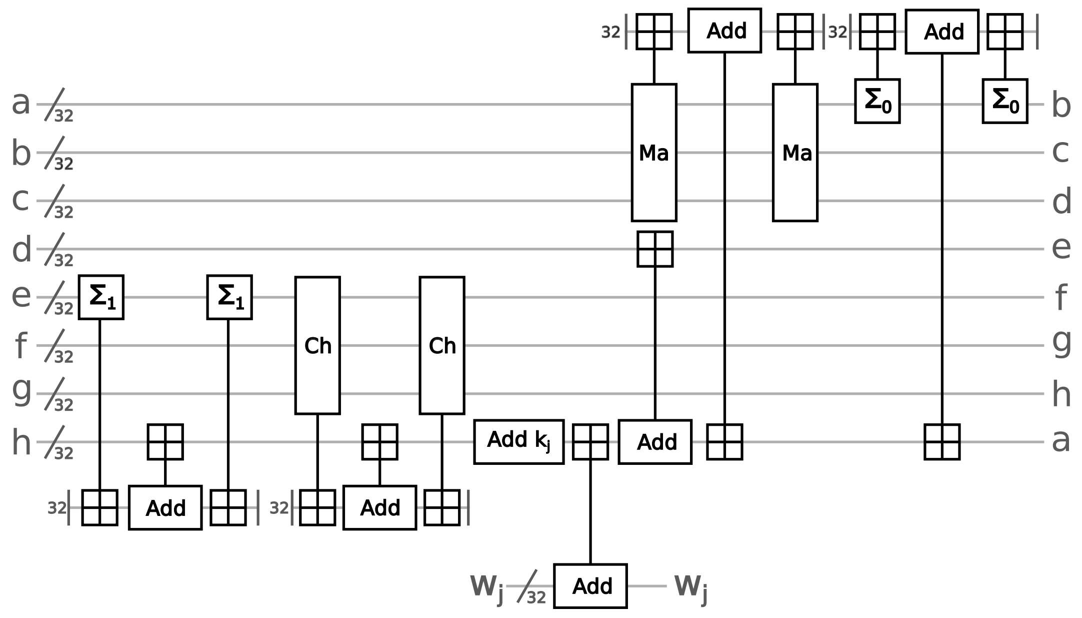


Figure 2.8: One Round of SHA-256. Constructed using the Eager cleanup method.

2.3 Summary

Pebble games are a useful tool in the analysis of space-time reversible circuits. The reversible pebble game does not represent operations which modify registers without additional allocation (for example the CNOT operation: $(a, b) \mapsto (a, a \oplus b)$).

An alternative representation (the MDD game) where these operations can be represented is proposed. Using this representation is is possible to generate more space efficient circuits.

Chapter 3

Arithmetic Circuits

A library of commonly used arithmetic functions is needed for implementing a basic set of built in functions in a reversible compiler. Extra consideration is given to in-place functions whenever possible due to the improved efficiency provided by reversible mutation in MDD based cleanup.

This section also demonstrates the usefulness of pebble based circuit analysis. It is possible, using pebble games, to achieve an asymptotic improvement over previous results in the space-time product for reversible Karatsuba.

3.1 Addition

An addition circuit is a circuit which takes two n bit integers $\mathbf{a} = a_1 a_2 \dots a_n$ and $\mathbf{b} = b_1 b_2 \dots b_n$ and returns the sum $\mathbf{a} + \mathbf{b} \mod 2^n$. The operation $(a, b) \mapsto (a, a + b)$ is injective so it is expected that an addition circuit which is in-place on one of its input exists.

3.1.1 Classic Ripple

The classic ripple addition method simply adds the bits in each column starting from the least significant bit. The output for that bit is the result modulo 2 and a carry is set to be added to the next column if the result is > 1.

To perform this algorithm we will need a circuit which takes three bits then calculates the carry and sum. These can then be applied iteratively taking the two bits from each column of the numbers that are to be added as well as the previous carry.

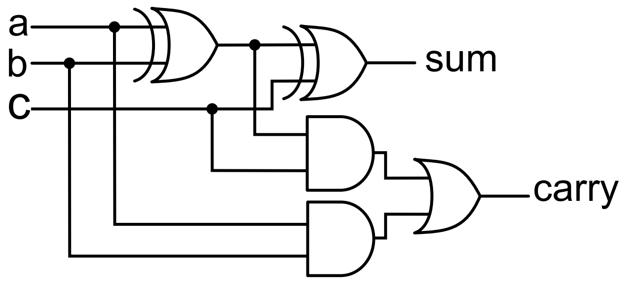


Figure 3.1: Irreversible full adder.

This can be done with a full adder circuit. A full adder is a circuit which takes input bits a, b, and c then returns the sum $(a \oplus b \oplus c)$ as well as the carry $(ab \oplus ac \oplus bc)$. Figure 3.1 shows an irreversible implementation.

In the reversible case a circuit (see figure 3.2) can be constructed that performs the mapping:

$$(a, b, c, 0) \mapsto (a, b, a \oplus b \oplus c, ab \oplus ac \oplus bc)$$

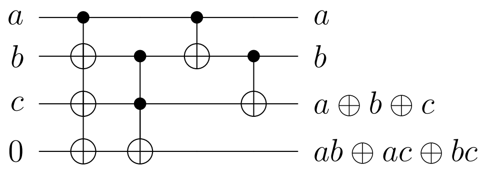


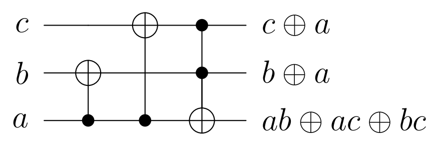
Figure 3.2: Reversible full adder [21]. Partially in place since c is overwritten.

These can then be chained together as they are in the irreversible case. The result will be a circuit which takes inputs (a, b) and produces outputs $(a, b, a + b)$. Note that since a Toffoli gate can be implemented in T-depth one [20] an out-off-place adder can be implemented in T-depth n where n is the bit width of its inputs.

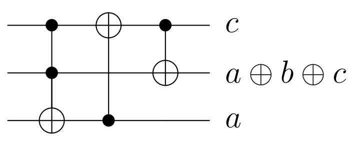
3.1.2 In-Place Ripple

It is sometimes the case that one of the input values is unneeded after the operation. In that case it would save space to overwrite the value. To do this we could try to construct an adder which implements the injective map $(a, b) \mapsto (a, a + b)$.

Such an adder is described Cuccaro et. al. in [6] (Note that not all the optimizations described in the original paper are desirable with our gate set as we wish to maximize shared controls). It is very similar to the classic ripple, the main improvement is the realization that information about the carry could be stored in one of the input bits of each column. This allows it to overwrite one of its inputs with the resulting sum. The circuit uses two main operations: MAJ, and UMA (See fig. 3.3). The MAJ operation takes an input carry c and two input bits a and b. It computes the carry onto a and partially computes the sum on b, it leaves c in an unclean state to be fixed later. This circuit can be repeated and rippled through all of the columns. After that is done the UMA operation can be performed. This operation cleans up a and c then finishes computing the sum on b.



(a) MAJ



(b) UMA

Figure 3.3: Basic operations used in the adder.

The cost of this adder is $2n-2$ Toffoli gates for an adder $\mod 2^n$. Note that each Toffoli gate has two shared controls which means it is possible to cancel at least two T-gates per Toffoli.

3.1.3 Controlled Ripple

This is a simple Controlled addition circuit with a gate count and depth $\Theta(n)$. Note that not all gates need be controlled; controlling a set of gates which if removed would transform the circuit into the identity is sufficient[1]. In the case of the in-place adder the MAJ and UMA subcircuits can be made to cancel by removing one gate each. Figure 3.4 shows the resulting circuit.

[1]Let U_{ctrl} denote a controlled U operation. The simplest example of this is $W = UVU^{-1}$. Here we can construct $W_{\text{ctrl}} = UV_{\text{ctrl}}U^{-1}$. As a slightly more complex example say $ABC = I$ and $X = AYBZC$ then $X_{\text{ctrl}} = AY_{\text{ctrl}}BZ_{\text{ctrl}}$.

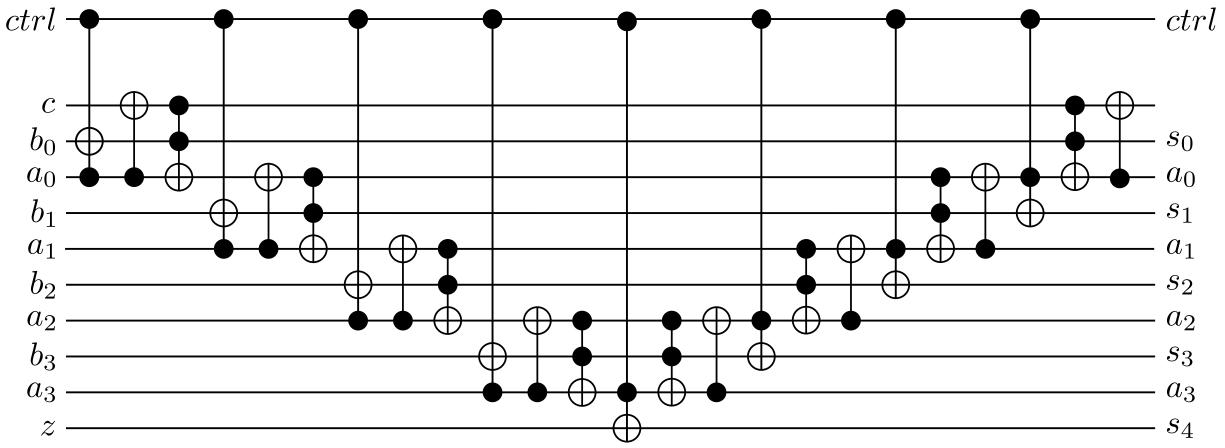


Figure 3.4: Controlled In-Place Ripple Adder. Based on the adder from [6].

3.1.4 Analysis

This modification can be accomplished by changing $2n + 1$ CNOT gates into Toffoli gates. This gives a controlled addition circuit of size n (A_n^{ctrl}) a total gate count:

$$A_n^{ctrl} = 4n + 1 \tag{3.1}$$

3.2 Multiplication

3.2.1 Controlled Addition Multiplier

A very simple implementation of multiplication, which is described in [17] uses controlled addition circuits (see fig. 3.4). Given two numbers as bit strings a and b their product can be found by repeatedly shifting forward by one and adding b to the result controlled on the next bit in a. See fig. 3.5 for a four bit example. This is essentially the elementary school shift and add method of multiplication on binary numbers. It is out of place and uses only one additional ancilla for the adder circuits.

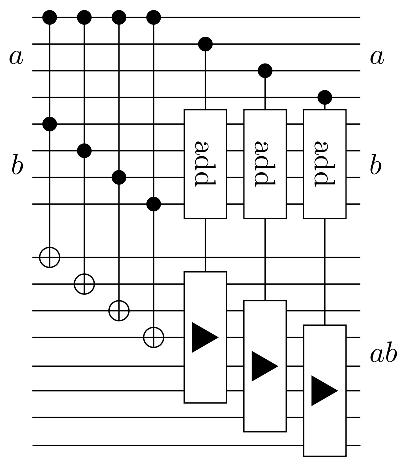


Figure 3.5: Controlled Addition Multiplier. Triangles are used in this figure to show which bits are modified by the adder.

Analysis

This circuit takes n Toffoli gates to copy down the initial value. It then uses $n-1$ controlled in place addition circuits to produce the final value.

So if we define A_n^{ctrl} to be the Toffoli count for a controlled adder of size n we get $M_n = n + (n-1)A_n^{ctrl}$ Where M_n is the gate count for a controlled addition based multiplication circuit of size n. From (3.1) we know the controlled addition circuit uses $4n+1$ Toffoli gates this gives a full count:

$$M_n = 4n^2 - 2n - 1 \tag{3.2}$$

3.2.2 Reversible Karatsuba

Let $n \geq 1$ and let x and y be n-bit integers. The Karatsuba [10] algorithm is based on the observation that by writing $x = x_1 2^{\lceil n/2 \rceil} + x_0$ and $y = y_1 2^{\lceil n/2 \rceil} + y_0$ the product xy can be evaluated as $xy = 2^n A + 2^{\rceil n/2 \rceil} B + C$, where

The following reversible algorithm for Karatsuba improves upon previous work [19, 11, 17]. An asymptotic improvement is space use (yielding as well an asymptotic improvement in the space-time product), is shown by using pebble games in the analysis. Further some constant improvements are realized by using in place addition to minimize garbage growth at each level and by optimally splitting the input in each recursive step rather then just dividing the bit-string in half, this is helpful when the integer size is not a power of 2.

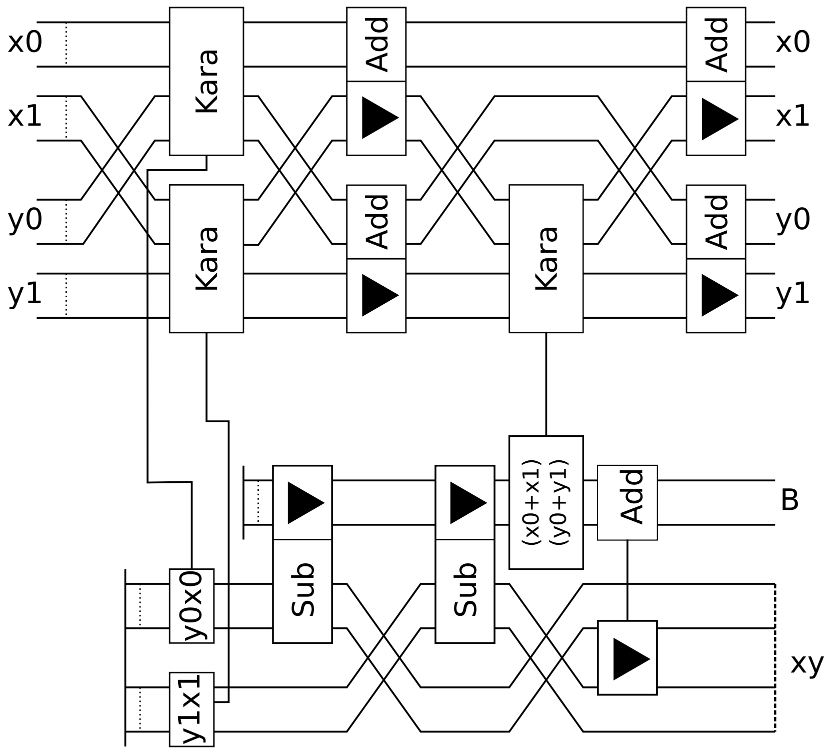


Figure 3.6: Karatsuba Multiplication Circuit. Triangles are used in this figure to show which bits are modified by the adder.

$$\begin{aligned} A &= x_1 y_1, \\ B &= (x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1, \\ C &= x_0 y_0. \end{aligned}$$

Analysis

Note that the cost for the computation of A, B, and C are 3 multiplications and four additions. Note further that the additions to compose the final result do not have to be carried out as the bit representation of xy is the concatenation of the bit representations of

A, B, and C. For $m \geq 1$, let M_m^g denote the Toffoli cost of a circuit that multiplies m-bit inputs x and y using ancillas, i.e., a circuit that maps $(x, y, 0, 0) \mapsto (x, y, g(x, y), xy)$, where xy is a $2m$-bit output, and $g(x, y)$ is an garbage output on $k \geq 1$ bits. Furthermore, denote by A_m the cost for an (in-place) adder of two m-bit numbers. It is known that A_m can be bounded by at most $2m$ Toffoli gates. Let K_n denote the number of Toffoli gates that arise in the quantum Karatsuba algorithm (See fig fig. 3.6). The outputs of one step of the recursion are x_0,x_1, y_0, y_1, x_0y_0, x_1y_1, and xy. It is easy to see that allowing garbage, K_n^g can be implemented using three multipliers of half the bit size, three in-place adders of size n and four in place adders of size $n/2$ (note the subtracters are just reversed adders). The base case is a multiplier for two one-bit numbers which can be done with one Toffoli gate, i.e., $K_1^g = 1$. We obtain the following recursion:

$$K_n^g = 3K_{n/2}^g + 5A_n; \quad K_1^g = 1. \tag{3.3}$$

For the overall clean implementation of the Karatsuba algorithm we first run this circuit forward, copy out the final result using n CNOTs, and then run the whole circuit backward. This leads to an overall cost of $K_n = 2K_n^g$ and n CNOTs. For the moment we focus on the Toffoli cost only. By expansion we obtain that:

$$\begin{aligned} K_n^g &= 3^{\log_2(n)} K_1^g + 5A_n + 15A_{n/2} + \ldots + 5 \cdot 3^{\log_2(n)-1} A_2 \\ &= 3^{\log_2(n)} K_1^g + \sum_{i=0}^{\log_2 n - 1} 3^i 5 A_{n/2^i} \end{aligned} \tag{3.4}$$

Using that the Toffoli cost of $A_{n/2^i}$ is $2n/2^i$, we obtain for the overall Toffoli cost the following bound:

$$\begin{aligned} K_n &= 2\left(3^{\log_2 n} + 10n \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i\right) \\ &= 2n^{\log_2 3} + 20n\left(\frac{1 - (3/2)^{\log_2 n}}{1 - 3/2}\right) \\ &= 2n^{\log_2 3} + 40n\left((3/2)^{\log_2 n} - 1\right) \leq 42n^{\log_2 3} \end{aligned} \tag{3.5}$$

This bound can be improved by replacing the recursive call to Karatsuba with naive multiplication once a certain cut-off has been reached (i.e. once n goes below some cutoff value). Looking at fig. 3.12 we see a comparison of various cutoff values (the naive method is also plotted for reference).

Another way to improve this algorithm is to attempt to choose more intelligent splits rather then always splitting the inputs in half at each level. This is important because the bit length of the numbers we are adding together may not be a power of two so dividing the input bit-string in half at each level might not be optimal. In fig. 3.12 the line plotted as `AKara10` shows the result of using the optimal splits at each level. These were found by a simple dynamic program which evaluated the total gate size for every possible split at every level and chose the optimal ones.

As using these methods we find an optimal cutoff value of 11 (see fig. 3.7).

Time-Space Trade-offs

We see in figs. 3.12 and 3.13 that there are trade-offs available between circuits size and gate count available by changing the cutoff value. A higher cut-off value value results in a larger naive multiplication circuits which are much more space efficient.

The reversible pebble game may be used to gain an asymptotic improvement in the space required to implement this algorithm. Note the tree structure of the recursive dependencies shown in fig. 3.11. We find a level k such that the size of each nodes subtree is approximately equal to the size of all nodes at that level and above (as shown in fig. 3.9). Then for each node at that level in sequence compute the node and uncompute all nodes below it. Once all the nodes are computed compute the rest of the tree. This will result in an asymptotic improvement in the amount of space used while only applying a constant multiplier (< 2) to the overall time.

For Karatsuba circuit on input of size n at a level x in the tree there are 3^x nodes of size $2^{-x}n$ for a total cost of

$$n\left(\frac{3}{2}\right)^x.$$

So the total cost of the full tree is given by

$$n\sum_{i=0}^{N}\left(\frac{3}{2}\right)^i,$$

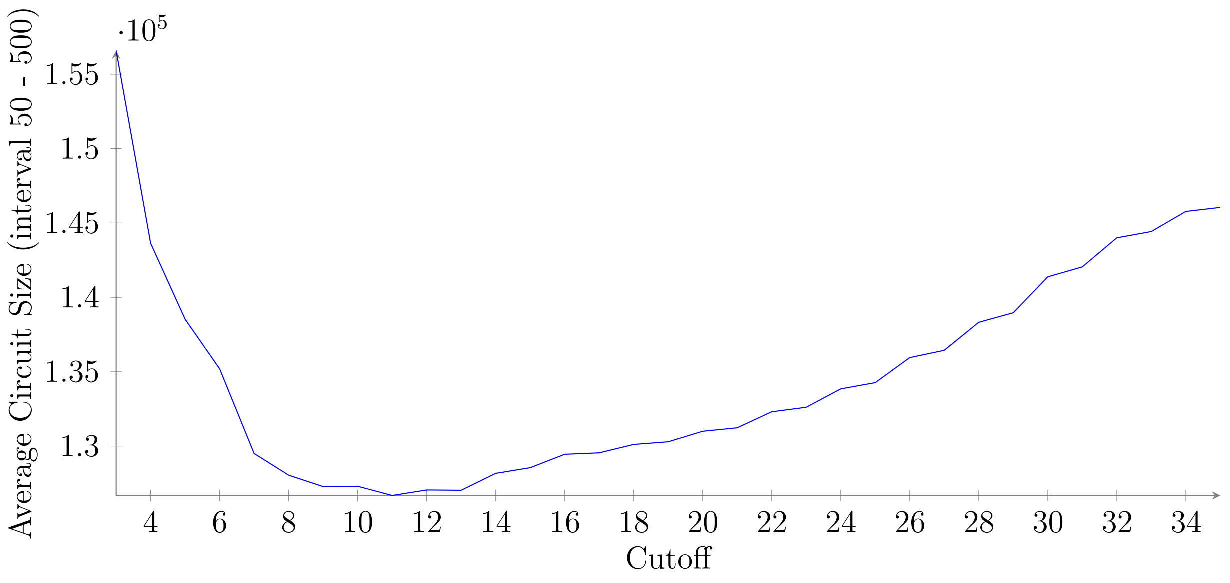


Figure 3.7: Average circuit size over the interval 50-500 for various cutoff values.

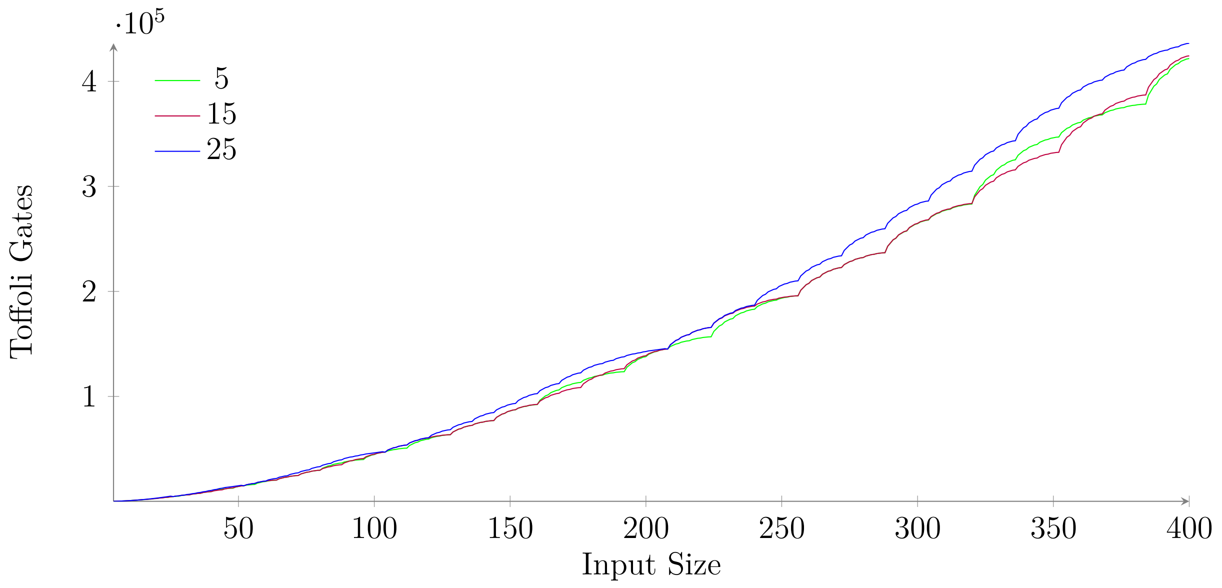


Figure 3.8: Comparison of various cutoffs for the adaptive cutoff version.

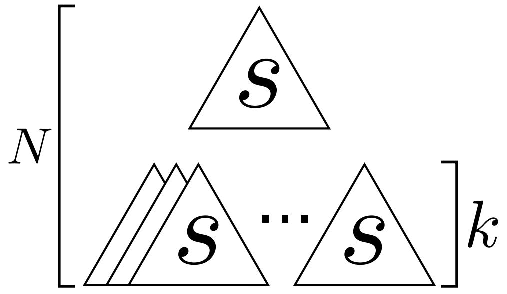


Figure 3.9: Partitioning of the Karatsuba tree

where $N = \log_2 n$.

We would like to break this tree into approximately equal sized subtrees at some level. Each tree at that level will be computed then uncomputed leaving only the top node. To minimize space we will choose the size of these subtrees to be approximately equal to the remaining size of the tree above them. To find the height k of such a tree we set:

$$\sum_{i=0}^{N-k-1} \left(\frac{3}{2}\right)^i = \frac{1}{2^{N-k}} \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i .$$

Since this is a geometric series we can use the identity $\sum_{k=0}^{n-1} r^k = \frac{1-r^n}{1-r}$ which holds for all r and obtain

$$\frac{1 - {}^{3}\!/_{2}{}^{N-k}}{1 - {}^{3}\!/_{2}} = \frac{1}{2^{N-k}} \frac{1 - {}^{3}\!/_{2}{}^{k}}{1 - {}^{3}\!/_{2}}.$$

Rearranging terms, we obtain

$$1 - {}^{3}\!/_{2}{}^{N-k} = 2^{k-N} - \frac{3^k}{2^N}.$$

Since $k \leq N$ and since we want that ${}^{3}\!/_{2}{}^{N-k} \geq \frac{3^k}{2^N}$ a simple calculation shows that this will be the case for $k \leq \frac{N}{2 - \frac{\log 2}{\log 3}} = 0.731N$. The total space use without this optimization is normally calculated as

$$n \sum_{k=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^k = n \frac{1 - (3/2)^{\log_2 n}}{1 - 3/2}.$$

Which gives space use of $O(n(3/2)^{\log_2 n})$ which is equivalent to $O(n^{\log_2 3})$ or approximately $O(n^{1.585})$ Using the above optimization we get space usage that can be bounded by

$$O(n(3/2)^{\frac{\log 3}{2\log 3 - \log 2}\log_2 n}) \approx O(n^{1.428}).$$

To find the depth of the circuit note that each node at level k must be computed sequentially. At level k the number of trees is

$$3^{\left(1 - \frac{\log 3}{2\log 3 - \log 2}\right)\log_2 n}.$$

Each tree is of depth :

$$\frac{n}{2^{1 - \frac{\log 3}{2\log 3 - \log 2}}}$$

This gives an overall depth for computing the k level of:

$$n\left(\frac{3}{2}\right)^{\left(1 - \frac{\log 3}{2\log 3 - \log 2}\right)\log_2 n} \approx n^{1.152}$$

This gives a space-depth product in the fully parallel version of $n^{1+\log_2 3}$ which is the same

Note that we have only used the reversible pebble game in this analysis. The MDD for our implementation of Karatsuba is shown in fig. 3.10. Notice that each of the recursive calls to $K_{n/2}$ can be preformed individually. This allows a further constant factor reduction is space use to be gained since we can now split each of the bottom subtrees into three parts. Each of these parts can be computed and uncomputed individually without increasing the overall time complexity.

3.2.3 Generalization to other recursions

Assume that we are given a function with input size n which splits a problem into a subproblems of size n/b where total work done to subdivide is $O(n)$. Then the overall work to compute the function for a problem of size n is given by:

$$\begin{tikzcd}
 & x, y \arrow[dl, dashed] \arrow[dr, dashed] \arrow[rr, dashed] & & x_0 + x_1, y_0 + y_1 \arrow[dr, dashed] & & B \arrow[dr, dashed] & \\
K_{n/2} \arrow[rr] & & K_{n/2} \arrow[rr] & & K_{n/2} \arrow[rr] \arrow[ur, dashed] & & K_n
\end{tikzcd}$$

Figure 3.10: MDD for the Karatsuba Multiplication Circuit.

$$n \sum_{i=0}^{N} \left(\frac{a}{b}\right)^i .$$

Solving as above we have:

$$k \leq \frac{\log_b n}{2 - \frac{\log b}{\log a}} .$$

This means that our method is effective for recursive functions where the number of sub-problems is greater than the problem size reduction factor. This is intuitive since if the problem size reduction factor is equal to the number of sub-problems then the sum over the space requirements of each node in a given level will always be equal. So the strategy discussed above would only give a constant factor space reduction.

By setting b in $\log b / \log a$ equal to one we get a square root reduction in space. This should be compared with a pebble games for complete binary graphs that was reported on in [13] in which a similar recursive structure was considered.

3.3 Division

In fig. 3.15 is a simple integer division circuit based on the binary form of long division as outlined in algorithm 2.

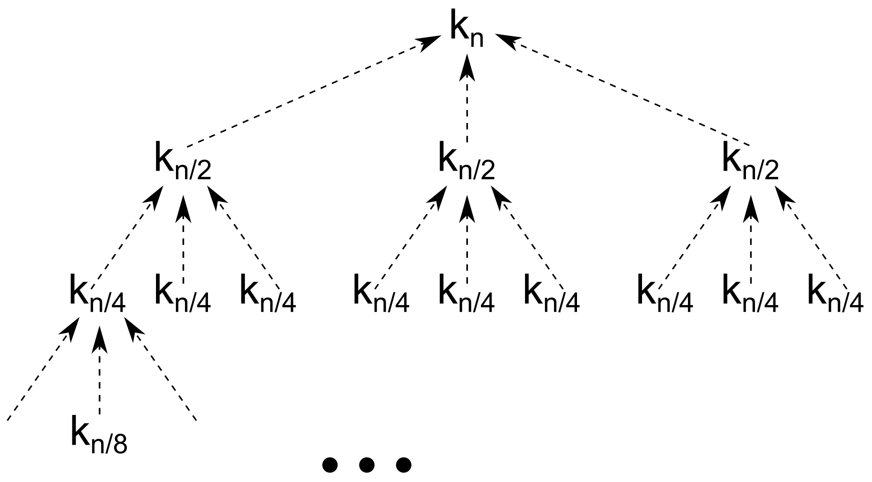


Figure 3.11: Structure of a pebble game for implementing the Karatsuba circuit. Note that K_n can be implemented using $n/2 + 1$ ancilla.

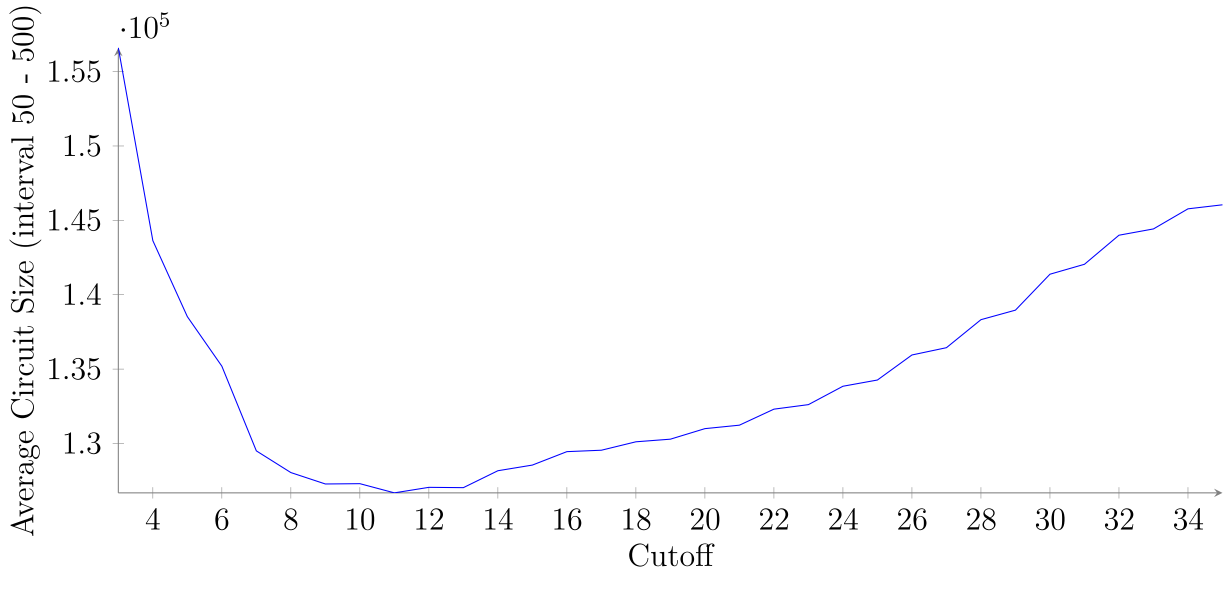


Figure 3.12: Plot of circuit sizes vs input size for various various Karatsuba cutoffs. The Legend shows the implementation (sKara for the simple version and aKara for the adaptive cutoff) as well as a number indicating the cutoff size.

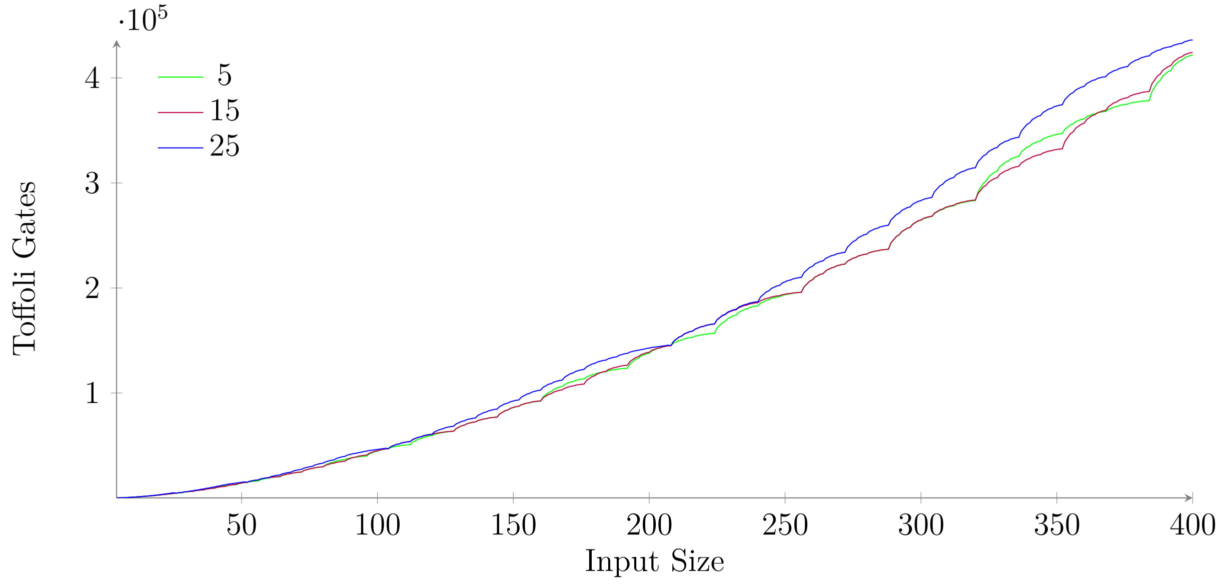


Figure 3.13: Plot of bits used verses input size for various Karatsuba cutoffs.

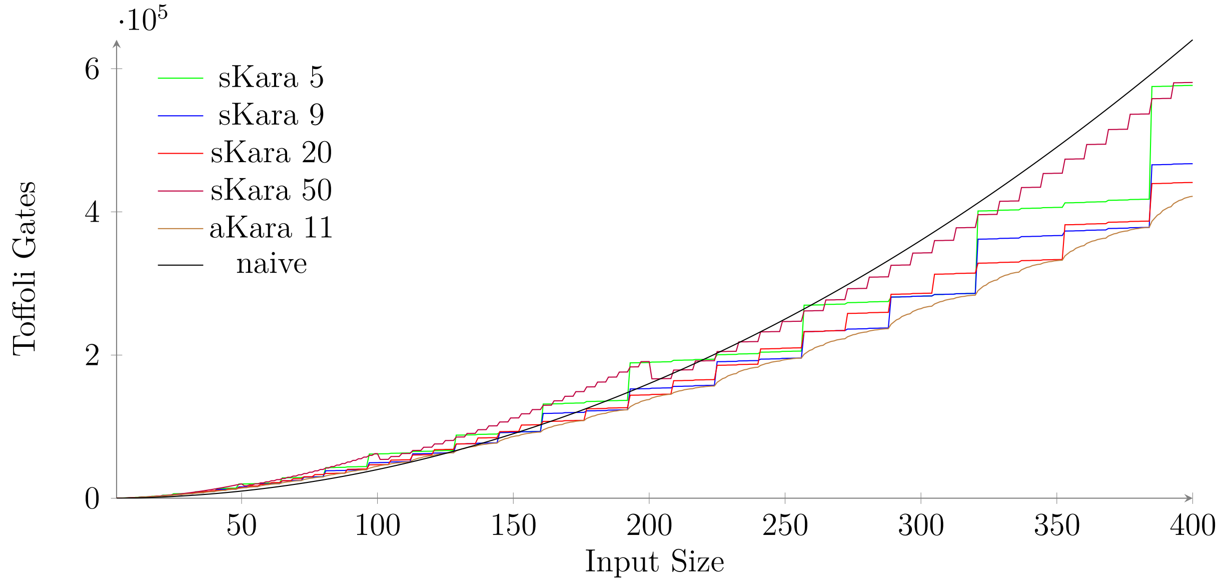


Figure 3.14: Plot of Toffoli depth verses input size for various Karatsuba cutoffs.

Algorithm 2: Integer Division with Remainder: find R and Q for N/D

```
Require: $D \neq 0$
 1: $Q \leftarrow 0$
 2: $R \leftarrow 0$
 3: for $i = n-1 \dots 0$ do
 4:     $R \leftarrow R \ll 1$
 5:     $R(0) \leftarrow N(i)$
 6:     if $R \geq D$ then
 7:         $R \leftarrow R - D$
 8:         $Q(i) \leftarrow 1$
 9:     end if
10: end for
```

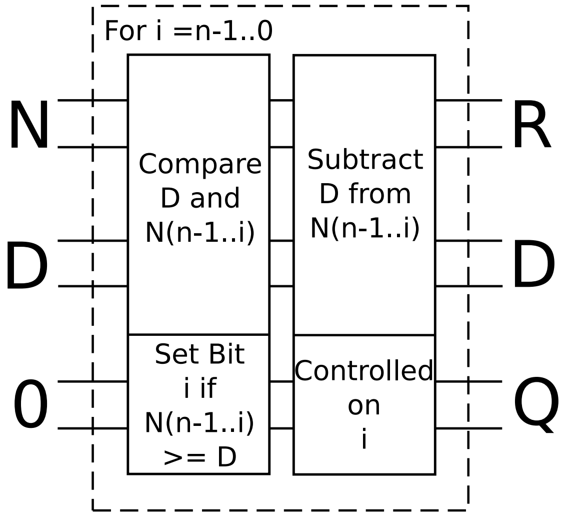


Figure 3.15: Binary Division Circuit.

An interesting feature of this algorithm is that combined with an in-place adder [6] it can easily be made partially in-place. On line 5 instead of allocating new space for $R(0)$ we can just use $N(i)$ as $R(0)$ since it will not be used again in the algorithm. In doing this we will overwrite the value of N with R as the algorithm is run. The if statement on line 6 can be performed with a comparison circuit. The result bit can be used to control the subtraction $R - D$. The value of the result bit is the same as $Q(i)$ after the assignment on line 8, so $Q(i)$ can just be taken to be the result bit. This avoids additional ancilla and the assignment to $Q(i)$.

This divider uses a total of n comparisons and subtractions of size n where n is the bit width of the input. A comparison can be implemented by computing the highest order bit when the numbers are subtracted.

3.4 Trigonometric functions with CORDIC

The CORDIC [22] algorithm was originally developed to calculate trigonometric functions on simple analog computers. Because of this the algorithms uses only simple operations such as bitshift, addition, and negation. Since these operations are simple to implement as reversible computations (and can in fact be done in-place) it is well suited for adaptation as a reversible algorithm. The basic idea of CORDIC is to apply repeated rotations which decrease in size with each iteration while using a comparator to choose the direction of each iteration such that they converge on the desired value (as shown in fig. 3.16):

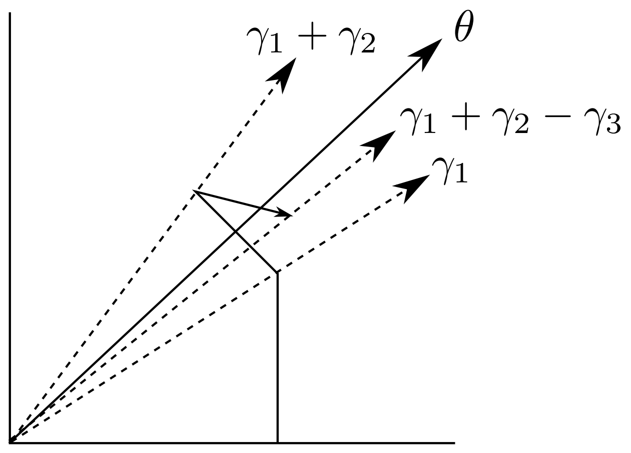


Figure 3.16: Rotations are used to converge on θ.

$$\theta = \alpha_0 + \alpha_1 + \cdots + \alpha_{n-1}$$

The issue is that computing rotations is generally expensive; requiring multiplication and the evaluation of trig functions (the very thing we are using it to accomplish!). An important realization of the CORDIC method is that it is possible to choose rotation angles such that this problem is avoided.

A rotation matrix is given by:

$$R_i = \begin{bmatrix} \cos\gamma_i & -\sin\gamma_i \\ \sin\gamma_i & \cos\gamma_i \end{bmatrix}$$

Using trig identities we can simplify this to:

$$R_i = K_i \begin{bmatrix} 1 & -\tan\gamma_i \\ \tan\gamma_i & 1 \end{bmatrix}$$

where $K_i = \frac{1}{\sqrt{1+\tan^2\gamma_i}}$.

We can collect all the K_i terms into a pre-computed scaling factor:

$$K(n) = \prod_{i=0}^{n-1} K_i$$

This can be applied with a single multiplication after the computation is complete.

We can select $\gamma_i = \tan^{-1} 2^{-i}$ as our set of angles so that all multiplications can be done simply as bit shifts. These angles will be pre-computed All that is left to do is find the directions of rotation. We can approximate some angle θ with the iterative process: $\theta_{i+1} = \theta_i + \sigma_i \gamma_i$. We choose the rotation direction σ by comparing θ_i with θ. If θ_i is greater then θ we rotate in the negative direction. If θ_i is less then then θ we rotate in the positive direction.

This iterative process can be stated as:

$$
\begin{aligned}
x_{i+1} &= x_i - \sigma_i y_i 2^{-i} \\
y_{i+1} &= y_i + \sigma_i x_i 2^{-i} \\
\theta_{i+1} &= \theta_i - \sigma_i \gamma_i
\end{aligned}
\tag{3.6}
$$

Reversible Implementation

The CORDIC algorithm can be implemented reversibly.

We pre-compute all of our rotation angles $(\gamma_0, \gamma_1, \ldots, \gamma_{n-1})$ as well as the scaling factor $K(n)$. Our input is the angle θ and our output is (x,y) where $\tan\theta = y/x$.

Finding the directions of rotation The directions of rotation can be determined using the sign bit of θ. At each step we copy out the sign bit of θ then add the angle γ_i if θ is negative or subtract if it is positive. At the end the copied out set of bits σ determine the directions of rotation. θ will approach zero so the higher order bits could possibly be cleaned up at the end of the step. Figure 3.17 shows a circuit for calculating σ.

Performing Rotations The actual rotation section is done using the iterative process above (eq. (3.6)). This circuit is shown in fig. 3.18.

Analysis

Convergence of the CORDIC algorithm requires $O(n)$ rotations where n is the number of bits of precision required. The cost of each rotation is also $O(n)$. This means the total cost of performing CORDIC is $O(n^2)$.

If we were to allocate memory according to the naive method this would also mean a space used of $O(n^2)$ Instead an MDD pebble game is used to optimize this circuit. The

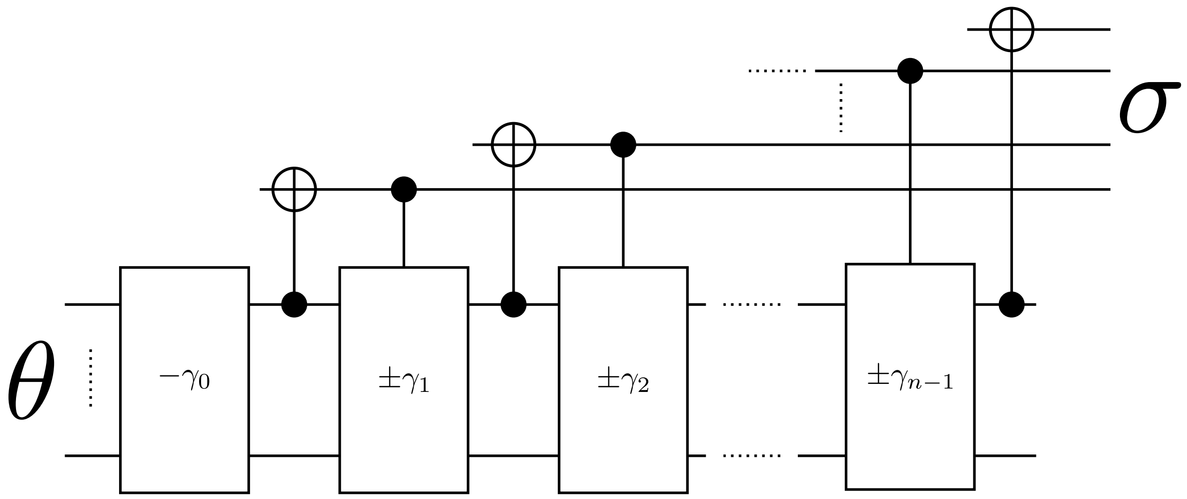


Figure 3.17: Circuit to determine directions of rotation. The controls determine whether γ is added or subtracted.

MDD for this implementation in shown in fig. 3.19. On the top part of the diagram are modification chains for x and y. The issue with these chains is that each new value is dependent on both of the previous values, i.e. $x_n + 1$ depends on x_n and y_n. This means that they cannot be both be advanced without leaving behind pebbles. We can choose to only leave pebbles behind on half the nodes, for example we can pebble $x_0, \ldots, X_n$ while advancing a single pebble along y. This however will only half the number of pebbles required.

Assuming we pebble the x chain and move the y pebble forward as described above we can see the pebbles along the x as a sort of 1D pebble game, we therefore can use the previously discussed strategies. With an incremental strategy we would only need to store $O(n^{1/2})$ x registers. This improves our overall space use to $O(n^{3/2})$ while leaving our time complexity unchanged at $O(n^2)$.

Figure 3.18 shows a possible generated circuit for the strategy where the x chain is fully covered while the y chain is advanced. As implied by the MDD this circuit can be partially parallelized with the circuit in fig. 3.17.

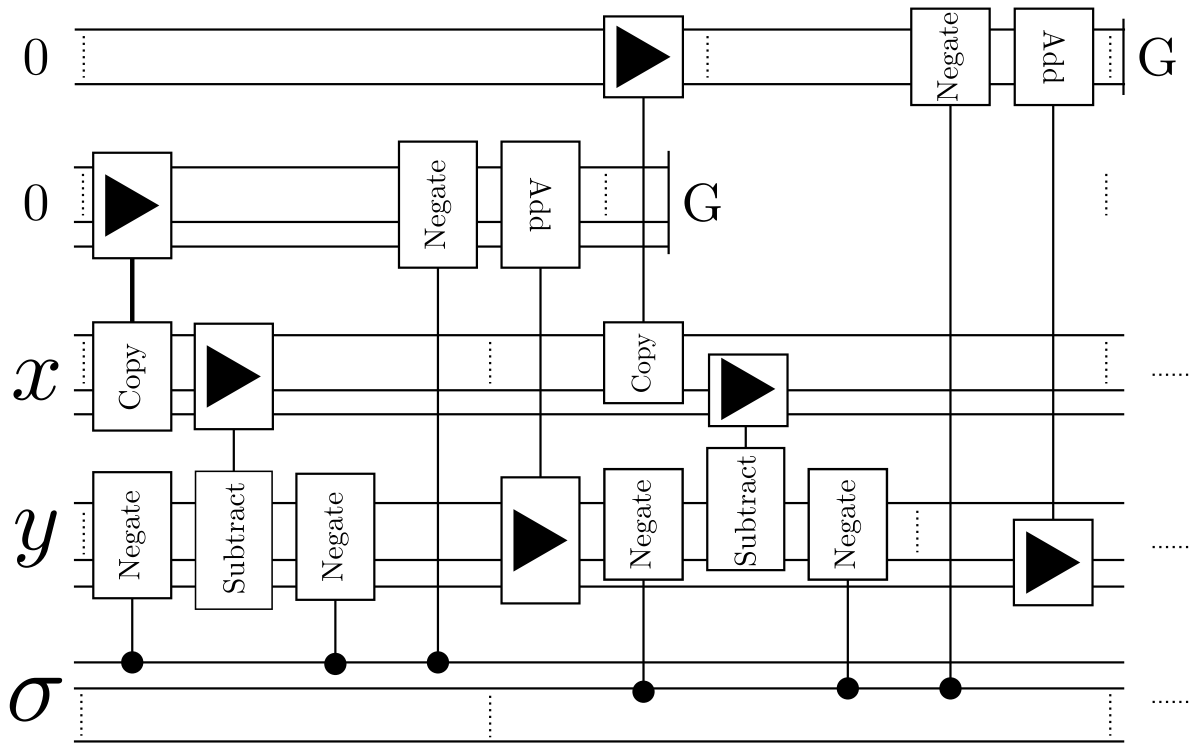


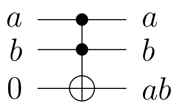
Figure 3.18: Circuit to perform rotations. x and y are both initialized to $1/\sqrt{2}$. Garbage is cleaned at the end by copying out the result and repeating the circuit.

$$\begin{tikzcd}
 & x_0 \arrow[rr] \arrow[drr, dashed] & & x_1 \arrow[rr] \arrow[drr, dashed] & & x_2 \arrow[rr] \arrow[drr, dashed] & & x_3 \arrow[r] \arrow[dr, dashed] & \cdots & & x_n \\
 & y_0 \arrow[rr] \arrow[urr, dashed] & & y_1 \arrow[rr] \arrow[urr, dashed] & & y_2 \arrow[rr] \arrow[urr, dashed] & & y_3 \arrow[r] \arrow[ur, dashed] & \cdots & & y_n \\
 & & \sigma_1 \arrow[uur, dashed] \arrow[ur, dashed] \arrow[drr, dashed] & & \sigma_2 \arrow[uur, dashed] \arrow[ur, dashed] \arrow[drr, dashed] & & \sigma_3 \arrow[uur, dashed] \arrow[ur, dashed] \arrow[drr, dashed] & & \cdots & \sigma_n \arrow[uur, dashed] \arrow[ur, dashed] & \\
\theta_0 \arrow[rr] & & \theta_1 \arrow[rr] \arrow[u, dashed] & & \theta_2 \arrow[rr] \arrow[u, dashed] & & \theta_3 \arrow[rr] \arrow[u, dashed] & & \cdots & \theta_n \arrow[u, dashed] &
\end{tikzcd}$$

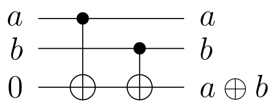
Figure 3.19: MDD for a reversible implementation of the CORDIC algorithm. Note that x_0 and y_0 are constants and θ_0 is the input to the function. The σ nodes are each one bit in size.

3.5 Boolean Expressions

AND operations can be reproduced using Toffoli gates onto ancilla:

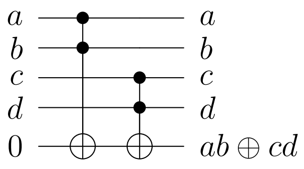


XOR operations can be reproduced using two CNOT gates onto ancilla:



Notice that AND operations are more costly both because they require a Toffoli gate rather then a CNOT gate, and because they can only be applied through a XOR operation rather then directly.

One simple optimization is to notice that the Toffoli gate includes an XOR operation. So we can calculate an expression such as $ab \oplus cd$ with:



One method of generating quantum circuits from boolean expressions is to first transform them into circuits of NOT, XOR, and AND operations. Given an expression with the recursive structure:

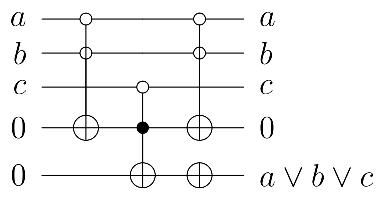
```
TERM = VAR
     | AND (TERM list)
     | XOR (TERM list)
     | NOT TERM
```

An algorithm to produce a circuit for such an expression is given by:

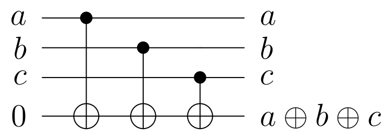
3.5.1 Converting OR to XOR

[18]In the case of mutually exclusive statements XOR is equivalent to OR. That is to say $a \vee b = a \oplus b$ if $a = 1 \implies b = 0$ and $b = 1 \implies a = 0$. For example $a \wedge b \vee \neg a \wedge c = a \wedge b \oplus \neg a \wedge c$.

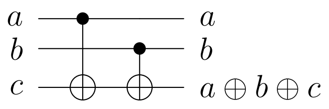
This is very useful as it allows us to avoid the use of Toffoli gates and use less ancilla. For example if we wished to compute $a \vee b \vee c$ we might use the circuit:



Where as $a \oplus b \oplus c$ can be computed as:



Or if one of a, b, or c are not needed in the later computation it can be done in-place as:



Given a set of AND expressions that are combined using OR we want to find sets of mutually exclusive statements that minimize the use of AND. We consider each AND expression to be a vertex on a graph and add edges between vertices that are mutually exclusive. Now we cover this graph using a minimum number of cliques.

After finding these cliques each set of mutually exclusive statements can be implemented by evaluating the AND statements and combining all of the values on a single ancilla using XOR for each clique. These results can then be combined using OR statements (since OR is associative it does not matter how we group the statements).

3.6 Gray Code Controlled Counter

Suppose you want to perform a set of functions which return some boolean value and you wish to know the first such function which returns 0. For example if we have an n-bit number and we wish to find the position of the first 0 value.

In this case we want a controlled increment function that stop incrementing the first time the control is 0. We therefore want to change the state of the counter by looking at just enough information to verify that all previous steps have been performed. To do this we must look at a subset whose value is unique to the previous step. Note some difficulty is introduced by the reversible nature of the circuit in that we cannot depend on parts of the state that we are updating. To do this we simply follow a binary counting method which flips one bit each step (Gray Code). Since only one bit is flipped per iteration the context of the unflipped bits is sufficient to verify the previous steps. An example of such a chain is shown below where the underlined portions of the state are the ones verified before the next step:

$$000 \rightarrow 00\underline{1} \rightarrow 0\underline{1}1 \rightarrow 0\underline{10} \rightarrow \underline{1}10 \rightarrow \underline{1}1\underline{1} \rightarrow \underline{10}1 \rightarrow 100$$

Chapter 4

Discussion and Open Problems

A new game for Heuristic methods for playing pebble games on an MDD were presented and pebble game analysis was applied to some arithmetic circuits of interest resulting in an asymptotic space reduction for a class of recursive functions (see 3.2.3). It is likely that there exist efficiently computable methods with better time and/or space performance.

Additionally methods for the conversion of other reversible languages to an MDD intermediate might give further insight into the types of useful heuristic strategies that might be implemented. More work should be done to explore MDD games on some special graphs which correspond to common computations.

It might also be interesting to create a sort of graphical reversible programming language where the programmer directly interacts with and constructs an MDD. Such a language might make it easier to visualize the structure imposed by reversibility.

The MDD structure may also be useful as a compiler intermediate for reversible languages since it seems general enough to be language agnostic and many optimizations and time-space trade-offs can be done using the information provided. This would allow optimizations provided by a good MDD $\mapsto$ circuit compiler to be shared between languages.

References

[1] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3. *arXiv preprint arXiv:1603.09383*, 2016.

[2] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013.

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

[4] C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18:766–776, 1989.

[5] Siu Man Chan. *Pebble games and complexity.* PhD thesis, University of California, Berkeley, 2013.

[6] S. A. Cuccaro, Th. G Draper, S. A. Kutin, and D. P. Moulton. A new quantum ripple-carry addition circuit. 2004. arXiv preprint quant-ph/0410184.

[7] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, 2012.

[8] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying grovers algorithm to aes: quantum resource estimates. In *International Workshop on Post-Quantum Cryptography*, pages 29–43. Springer, 2016.

[9] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *Journal of the ACM (JACM)*, 24(2):332–337, 1977.

[10] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[11] Shane Kepley and Rainer Steinwandt. Quantum circuits for $\mathbb{F}_{2^n}$-multiplication with subquadratic gate count. *Quantum Information Processing*, 14(7):2373–2386, 2015.

[12] E. Knill. An analysis of Bennett's pebble game. arXiv.org preprint quant-ph/9508218.

[13] Balagopal Komarath, Jayalal Sarma, and Saurabh Sawlani. Pebbling meets coloring: Reversible pebble game on trees. *arXiv preprint arXiv:1604.05510*, 2016.

[14] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961.

[15] Yves Lecerf. Logique mathematique-machines de turing reversibles-recursive insolubilite en $n \in n$ de lequation $u = \theta^n u$, ou *theta* est un "isomorphisme de codes". *Comptes Rendus Hebdomadaires des Seances de L'academie des Sciences*, 257(18):2597, 1963.

[16] Christopher Lutz and Howard Derby. Janus: a time-reversible language. *Caltech class project*, 1982.

[17] Sebastian Offermann, Robert Wille, Gerhard W Dueck, and Rolf Drechsler. Synthesizing multiplier in reversible logic. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pages 335–340. IEEE, 2010.

[18] Alex Parent, Martin Roetteler, and Krysta M Svore. Reversible circuit compilation with space constraints. *arXiv preprint arXiv:1510.00377*, 2015.

[19] R Portugal and CMH Figueiredo. Reversible karatsuba's algorithm. *Journal of Universal Computer Science*, 12(5):499–511, 2006.

[20] Peter Selinger. Quantum circuits of t-depth one. *Physical Review A*, 87(4):042302, 2013.

[21] Yvan Van Rentergem and Alexis De Vos. Optimal design of a reversible full adder. *International Journal of Unconventional Computing*, 1(4):339, 2005.

[22] Jack E Volder. The cordic trigonometric computing technique. *Electronic Computers, IRE Transactions on*, (3):330–334, 1959.

[23] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Proc. Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'07)*, pages 144–153. ACM, 2007.

Appendix A

REVS

REVS is a reversible programming language embedded in F#. It uses MDD based compilation with a simplified version of the eager cleanup strategy. The example below computes the main rounds function in SHA-256. The output of REVS is a circuit similar to the one shown in fig. 2.8. For a full explanation see [18].

```
for i in 0 .. n - 1 do
    h <- addMod2_32 (ch e f g)
    h <- addMod2_32 (s0 a)
    h <- addMod2_32 w
    h <- addMod2_32 k
    d <- addMod2_32 h
    h <- addMod2_32 (ma a b c)
    h <- addMod2_32 (s1 e)
    let t = h
    h <- g; g <- f; f <- e; e <- d
    d <- c; c <- b; b <- a; a <- t
```

Appendix B

jcc

rcc (Reversible Circuit Compiler) is a compiler implemented for a modified version of the Janus programing language [23]. It's code is available at https://github.com/aparent/rcc. It does not implement the full MDD scheme but does provide the equivalent to eager cleanup at the statement level.

The compiler targets quantum circuits.

B.1 Implementation

B.1.1 Gate Set

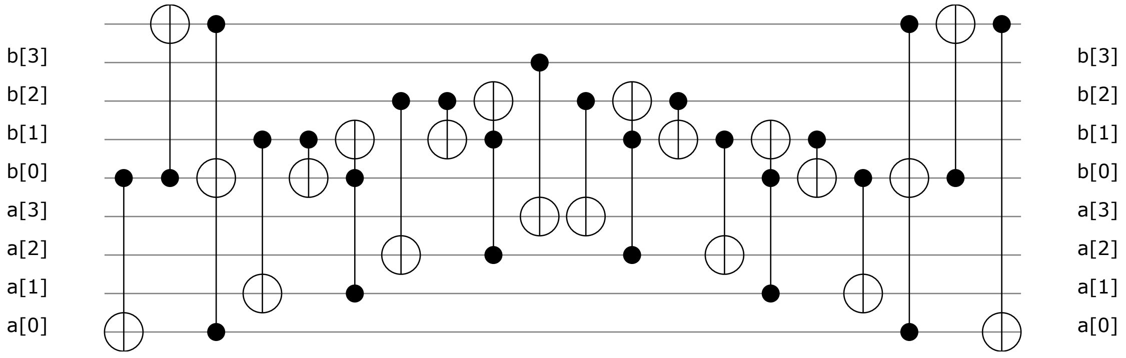
The gate set used by the compiler includes the NOT, CNOT, and Toffoli gates.

When choosing the implementation of various operations consideration is given to the expansion into the Clifford+T gate set. More specifically to the minimization of T-gates. For example shared controls on Toffoli gates are desired as they result in T cancellation.

B.1.2 Addition

Addition is done using the CDKM[6] adder as shown below:

```
a b;
a += b;
```



The final carry bit is not computed in this adder. The result is an adder which computes $a + b \mod 2^n$

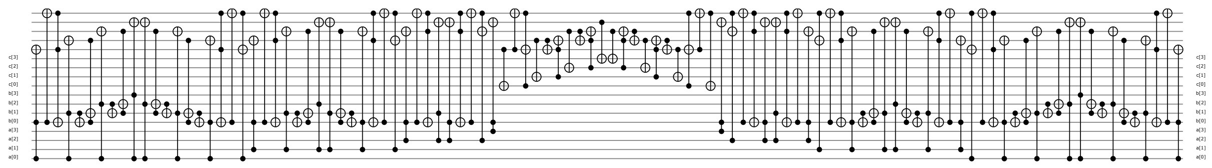
B.1.3 Subtraction

Subtraction can be done simply by reversing the addition circuit.

B.1.4 Multiplication

Multiplication is done with a simple shift and add circuit. An adder adding the variables a and b consists of a CDKM[6] adder controlled on each bit of a. Each adder adds b to some ancilla shifted left by the position of the control in a. Each adder is smaller then the last as the multiplication is performed $\mod 2^n$. Below is an example of the compiler output for a multiplication:

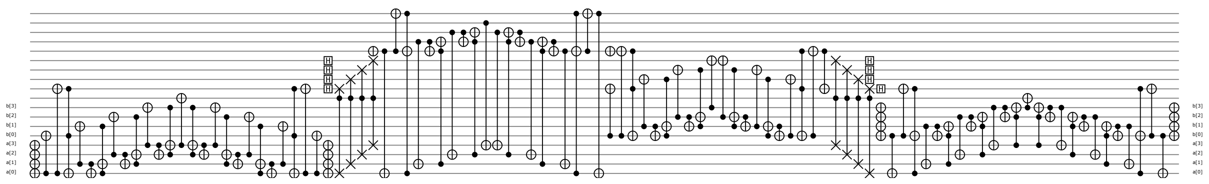
```
a b c;
c += a*b;
```



B.1.5 Conditional Statements

Conditional if-else branches can be evaluated by swapping the bits to the correct circuit path controlled on the if conditional. The other circuit path is evaluated on a set of ancilla bits is initialized to $H^{\otimes n}|0\rangle$. Since this is an eigenvector of every permutation matrix it will be unchanged and can be cleaned up by reversing the initialization.

```
a b;
if a < b then
  a += b;
else
  a += 1;
fi a > b;
```



Note that the assertion is a statement about the state of the program which is true if and only if the "if" branch is taken rather then the "else" branch. This is useful as it describes a property of our data that can be used to clean up the bit used to store the result of the conditional. i.e. If we know that we took that if branch rather then the else branch we can XOR the conditional bit and be assured it is being reset to zero.

B.1.6 Loops

In the circuit model we need to implement all loop operations up to some max bound. This is especially important in the Quantum model as some states in a superposition may require more loop iterations then others to evaluate.

So to implement a loop we want to repeatedly perform some computation until a condition is met. We then want to stop performing that computation for the remainder of the loop.

Swapping out of the loop using as done in conditional statements will not work since some of the superposition states will reset the condition bit.

On solution currently implemented in the compiler is to only allow loops for which the number of iterations is known at compile time. In that case it is simple to implement since the circuit will just be repeated a known number of times.

Alternatively if a conditional exit is required the counter described in section 3.6 can incremented conditionally on the exit condition. The body of the loop can then be controlled by checking against the expected value of the counter for that iteration.