Bachelor thesis
Computing Science

Radboud University

Linearly and arboreally stackable quantum-dot cellular automata and their discrete simulation

Towards a scalable 2^n-bit nano-scale processing cuboid

Author:
Willem Lambooy
willem.lambooij@ru.nl
s1009854

First supervisor/assessor:
dr. Cynthia Kop
C.Kop@cs.ru.nl

Second assessor:
dr. Sjaak Smetsers
S.Smetsers@ru.nl

August 20, 2021

Abstract

The contemporary semiconductor chip technology reduces transistor sizes at a rate that is asymptotic; a direct consequence of the increasing effect of the uncertainty principle that plays at the quantum scale. Exactly this effect enables the formation of quantum-dot cellular automata: a technology from the emerging nanocomputing paradigm that operates on the Coulombic repulsion between individual electrons.

In this thesis, we present *QCA-STACK*: a system written in the functional programming language Haskell that enables the discrete simulation of such automata – as opposed to continuous simulation with QCADesigner, the most widely used tool for this purpose since its conception in 2005. We show 2^n-bit designs of ALU components; designs contrived in such a way that a 2^n-bit component can be extended in the third dimension to produce a 2^{n+1}-bit component, essentially by stacking the design on top of itself. Furthermore, we present algorithms that perform such stacking operations for both linearly and arboreally stackable designs. With the aforementioned, we were able to discretely simulate a 64-bit RCA and a 32:1 multiplexer with consistent results.

Contents

Chapter 1

Introduction

All conventional electronic devices implementing digital logic circuits in use by consumers ranging from regular smartphone users to big corporations working on supercomputing solutions depend on the state of CMOS (Complementary Metal-Oxide-Semiconductor) technology and have done so since the late 1960s. The developments in this field predominantly revolve around sizing down building blocks used for integrated circuits, transistors specifically. Since its conception, the technology has improved at a rate very close to the one predicted by Gordon Moore in 1965 (Moore's Law). Or at least, until around a decade ago, that is.

Advancements made in the last decade show little hope for a future for Moore's projection and even Moore himself stated in 2005 that he expected his 'law' would cease to apply by 2020-2025. [2] As components are sized down to produce integrated systems in the order of nano-scale, physical boundaries are being reached and complications arise by cause of the increasing effect of the uncertainty principle that plays at the quantum-scale.

The era dominated by CMOS technology is coming to an end and with this, a new technology should arise. This new technology should not only improve over CMOS in terms of miniaturisation potential but also in terms of energy efficiency. Besides this, the technology should allow for the implementation of digital logic circuits that are similar to the ones its predecessor allows for, as this enables the many existing systems to be ported with relative ease.

Field-coupled nanocomputing (FCN) is a group of promising emergent technologies that satisfy the previously listed requirements. The technologies overcome issues relating to quantum effects that CMOS struggles with at its physical limits by playing on these quantum interactions directly and using them as a means of transducing a signal. FCN mimics and realises the concept of cellular automata, in which the state of a cell at a given time depends on the state of neighbouring cells at the preceding time. This has been shown to allow for the implementation of digital logic circuits. [15, 8, 19]

At the current time, FCN mainly exists as a theoretical concept due to the difficulty and cost of producing physical systems using contemporary technology. However, a great number of contributions have been made towards a physical realisation of the system, including but not limited to Molecular QCA, Atomic QCA, NanoMagnet Logic and Silicon Atomic QCA. [8, 29] All of these technologies have their advantages and disadvantages; NanoMagnet Logic can be produced using current-state technology and allows for 'cells' to be produced in sizes lower than 100 nanometre – while Silicon Atomic QCA, although more difficult to realise, enables an even further downscale to the order of single atoms and thus supports the production of single nanometre 'cells'. [19] The implementations all offer significant improvements over CMOS technology: a reduced scale by up to three orders of magnitude as previously discussed, and furthermore an extremely low power consumption and theoretical clock speeds in the order of Terahertz. [26, 21]

* * *

In this thesis, we will discuss one of the earliest of FCN technologies, proposed by Lent et al. in 1993: Quantum-dot Cellular Automata (QCA). [15] This technology is used as the overarching theoretical principle for all of the aforementioned physical FCN implementations and is therefore the focus of this work. This is a thesis in the field of theoretical computing science; we thus consider the system purely in a theoretical manner.

Even more so, in Chapter 3 we discuss how the physical system that emerges from the dynamics of quantum-dot cellular automata can be reduced to a functional system of rules that is consistent with the original system to an extent that is required for the purpose of simulation. Here we take the implementation found in a simulation engine of QCADesigner – the conventional QCA simulation solution – as our foundation, and transform this continuous simulation engine into a reduced discrete simulation engine to form QCA-STACK, a command-line QCA simulation and design tool written in Haskell.

Our functional take on the QCA system facilitates experimentation with the three-dimensional aspect of the QCA system. In Chapter 4, we present scalable designs of ALU components that exploit this three-dimensionality. Our method opens entirely new doors of scaling that could lead to the formation of an extensible, ultra-dense processing cuboid consisting of stackable components similar to towers – making it resemble the skyscraper-filled Downtown Manhattan. This novel way of scaling is realised through the extension of our system with functions that enable the generation of 2^n-bit designs by means of stacking operations: a 2^n-bit design is procedurally stacked on top of itself in the third dimension to produce a 2^{n+1}-bit design. Beside a linear stacking operation, we also present an arboreal one: this operation expands its input using the recursive quality of a tree structure.

Preliminarily, however, we discuss the foundational dynamics of QCA.

Chapter 2

Basic QCA dynamics

2.1 QCA Fundamentals

The scientific literature has varied ways of elaborating on this topic, differing slightly on terminology and depth of definition for certain concepts. In this chapter, a bespoke description of QCA dynamics is presented in which the logical functioning of the system is the key focus. This being a computing science thesis, physical phenomena are generally deemed extraneous to the focal point of this work and will thus be at most mentioned, though not expounded. The following publications were primary sources: [27, 30, 28, 3].

2.1.1 The cell

At the core of a quantum-dot cellular automaton is the QCA cell, a two-dimensional plane with equal length sides. Inside are four quantum-dots that are both spaced evenly apart from each other, and from the centre and the nearest corner of the cell. Each quantum-dot is a nanoscopic molecular structure that functions as a site that at most one electron can occupy. When exactly two electrons are put into a single cell, two stable states exist due to the repelling coulomb forces between the individual electrons. The two electrons will always be most stable in opposite corners, allowing us to distinguish the polarisations of the two stable states as $P = -1$ and $P = +1$ that can be read as binary `0` and binary `1` respectively.

$P = -1$ $P = +1$

Figure 2.1: The two stable states of a QCA cell.
Black indicates the presence of an electron in a quantum-dot.

When two cells are close to each other, this same force will cause the electrons to affect other nearby electrons through electro-static interaction. The cell walls prevent the electrons from escaping the cell, however. This property allows for the propagation of a signal as will be shown with the following example.

Take two directly adjacent cells, A and B. Cells can be fixed in a certain polarisation as will be discussed in the next subsection. Let this be the case for cell A and let its polarisation be $P = -1$. Since the state of this cell is locked, it acts as a driving force in the propagation of a signal; A is therefore called the driver cell in the current context. Cell B will follow and stabilise in the same polarisation, since e, the electron in cell A that is closest to cell B, repels the electrons in cell B from q, the quantum-dot in cell B that is closest to electron e. The electrons in cell B thus conform to the polarisation of cell A, making the polarisations of both cells identical. This property of signal propagation allows for the creation of logical gates in QCAs.

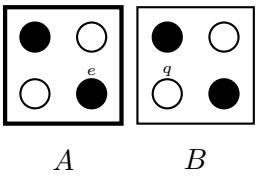


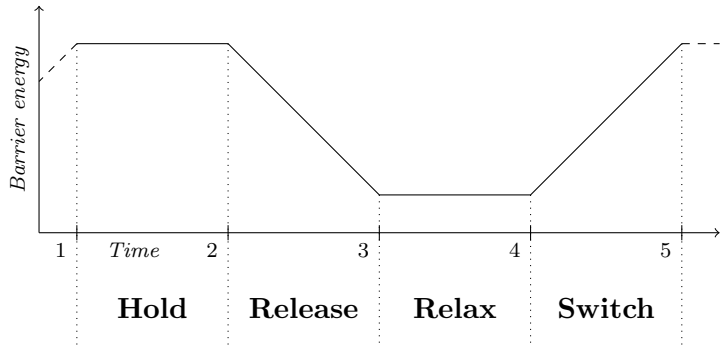
Figure 2.2: Cell B takes the polarisation of cell A.
The driver cell is indicated by the thick border.

2.1.2 The clock

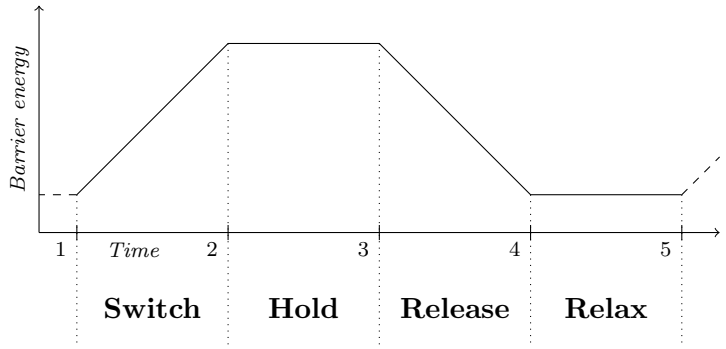
While it is possible for electrons to tunnel to nearby unoccupied quantum-dots in the same cell, they need an amount of energy for this that exceeds a certain tunneling threshold. This communal energetic property of the electrons in a cell is called the tunnelling energy (γ). To provide the means of locking a cell in a certain polarisation, i.e. restraining the electrons in this cells from tunnelling to other quantum-dots within the cell, each cell has intra-cellular barriers that govern the tunnelling threshold within the cell. If the barrier energy is high for this cell, we say that its barriers are raised; therewith the tunnelling threshold for the electrons within this cell is raised. This results in these electrons not having enough energy to tunnel to other quantum-dots, therefore fixing the polarisation of said cell, regardless of the electrostatic effects of neighbouring cells. Inversely, when the barrier energy is low, the barriers are lowered and the electrons are able to tunnel to other quantum-dots within the cell to conform to the least energetic configuration. It should be noted, however, that the polarisation of a cell is undefined when its intra-cellular barriers are lowered. Since the electrons can move freely in this state, no distinct polarisation can either be observed or propagated.

Synonymous with the barrier energy and therefore inversely proportional with the tunnelling energy is the clock signal. When the clock is high, the barrier energy is high and vice versa. This signal is useful for ensuring input signals are propagated correctly by applying a defined evaluation order of the cells. By making the clock high for some cell A from time $T = 1$ to time $T = 2$ while synchronously raising the clock from low to high for a neighbouring cell B, the latter cell will stabilise according to the polarisation of the former. Cell A is the driver cell here and finds itself in the Hold phase, as its polarisation is being held. Conversely, cell B is currently switching its polarisation from an undefined to a defined state and is therefore in the aptly named Switch phase. When this clock shift arrives at the upper bound at time $T = 2$ and the polarisation of cell B is now fixed, the signal has been propagated and the clock of cell A can be lowered.

The inherent cyclicity of the 'clock' concept applies to the QCA clock just as it applies to the clock found in CMOS architecture. The QCA clock cycles in a clipped sinusoidal manner where the clock-high and -low are the lower and upper clipping thresholds respectively. The four successive phases that emerge can be distinguished as follows: Hold (high clock), Release (high to low clock), Relax (low clock) and Switch (low to high clock).



(a) The four respective clock phases cell of A from the example.



(b) The four respective clock phases of cell B from the example.

Figure 2.3: The clock signal for cell A and cell B in section 2.1.2.

Continuing our example as shown in Figure 2.3, cell B can now propagate the signal further through the automaton. Take note that while we say that every cell has its own clock – in the same way that cell A and cell B have different clocks – these clocks are synchronised in their phase transitions and only differ in the starting clock phases. Consequently, multiple cells can have the same clock. At time $T = 3$, the clock of cell A will stay lowered for an additional time frame while its respective driver cells are polarised with a fresh input. This clock will then be incremented from the lower to the higher limit at time $T = 4$ to produce a distinct polarisation at time $T = 5$. At this time, the clock of cell A will be in the same state as it was when our example begun at time $T = 1$, although the actual polarisations of the cell at these times will be unrelated. One full clock cycle has been completed.

This example of signal propagation can be expanded to form a QCA wire, like the one shown in Figure 2.4. A connected array of cells with the same clock can propagate a signal just fine in many cases, yet multiple clock phases could be favoured for stability as will be touched on later.

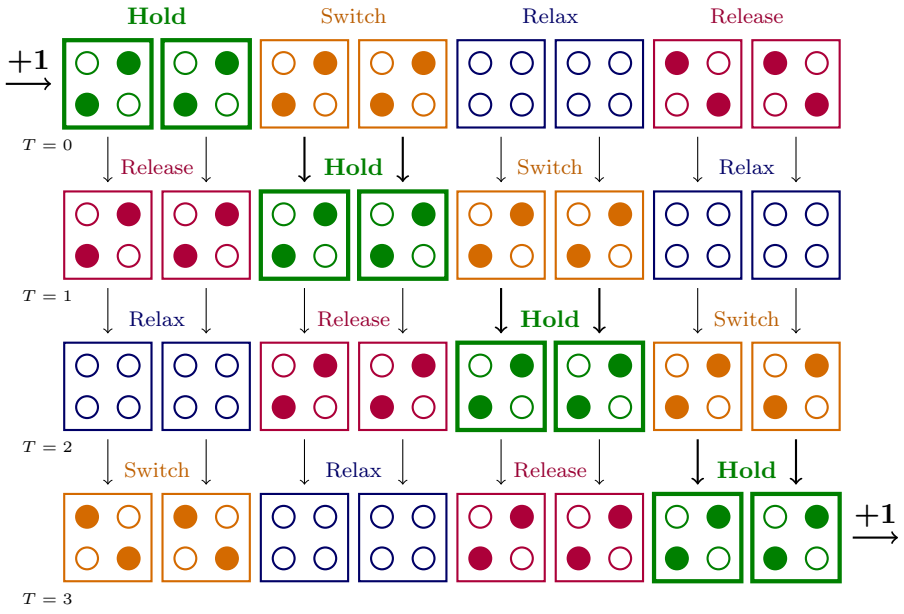


Figure 2.4: A QCA wire in four successive moments.
All consecutive instants displayed are one clock phase duration apart. The positive polarisation, or the signal $P = +1$, is carried from the left side to the right side of the wire.

An automaton continues cycling through the clock phases, applying new input values with every full cycle to each input cell when the cell enters the Switch phase, until it terminates when the final input has been fully propagated and each output cell has been read out in the Hold phase.

The astute reader might wonder why there is a need to assign entire phases to the high to low and low to high phase transitions. Why can the clock not be a two-state system with pseudo-instant state transitions like in CMOS? The literature presents a physical basis for this consideration, coupling it to the adiabatic theorem in quantum mechanics. [27] Appositely put, this theorem states that an abrupt change [in barrier energy] in a quantum mechanical system leads to the system having insufficient time to settle to the state of minimum energy. [1] Such non-adiabatic switching is not preferred for clock switches in QCAs, as proper control over a state following a clock switch is problematic.

Aside from physics, a more practical justification of the four-phased clocking signal proceeds from the extra stability of a QCA on a design level that it grants. Signal propagation with a QCA wire is not as straightforward as with connected two-state systems found in CMOS; longer wires require multiple clock phases for consistent signal transmission. Switching a cell in a wire formed of a longer array of same-clock cells in a cell-dense environment from an undefined to a defined polarisation can lead to a polarisation that is inconsistent with the wire's property of integrity, as seen in Figure 2.5. Each switching cell is sensitive to influence from any nearby cell in the Hold phase, so also from cells in a neighbouring wire. In a longer array of same-clock cells, the cells that are further away from the intended driver cells are more prone to accumulating the majority of incoming electro-static effects from unintended driver cells. With more equal-length clock phases, there is more space to prevent such interference when designing a QCA – albeit an increment in clock phases comes with the downside of reduced clock speeds.

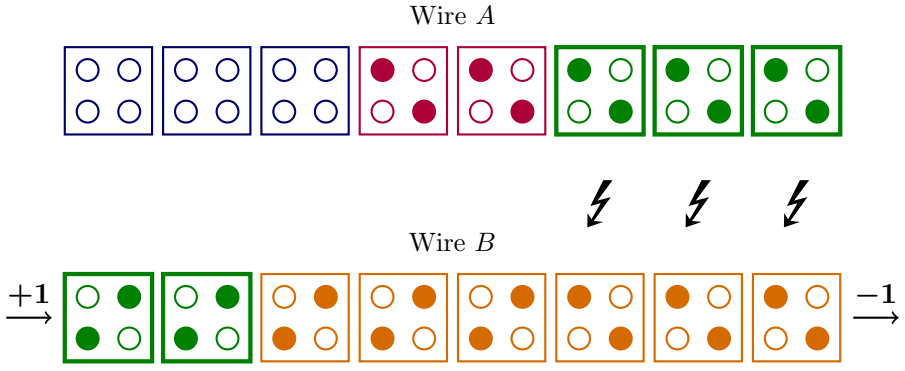


Figure 2.5: An example of a faulty wire.
Wire B does not retain the integrity of its signal ($P = +1$) since it is affected by cells in the Hold phase from neighbouring wire A. The swayed cells – the three rightmost cells displayed in wire B – are predominantly influenced by the driver cells in wire A, rather than by those in the wire they are a part of. These cells thus conform to the polarisation of the former set of driver cells.

2.1.3 Logical gates

Logical gates form the fundamental basis that enable digital logic. Any system that can implement and connect AND-, OR- and NOT-gates can in theory be used to forge sophisticated complexes, like a processing core.

There are two logical gates that are essential to QCA designs, and enable the creation of the three aforementioned gates. All figures in this section consist of driver cells (cells depicted with thick borders) and non-driver cells. The former can be regarded as cells in the Hold phase, whilst the latter as cells in the Switch phase.

The NOT-gate

The first one, the most elemental gate in every type of logic circuit, is the NOT-gate. This gate is very simply implemented by having the input cell be diagonally adjacent to the output cell so that two corners touch. In this configuration, electrons in the output cell will arrange themselves perpendicular to how the electrons in the input cell are arranged, inverting the polarisation of the output cell with respect to the input.

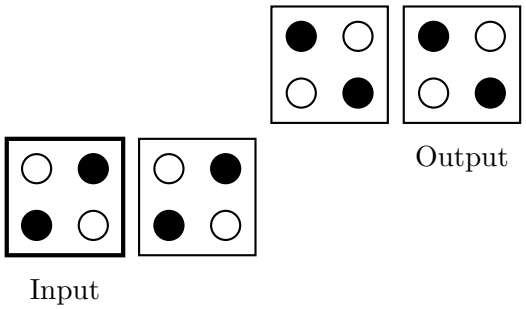


Figure 2.6: A NOT-gate.

The cause of this effect can be logically reasoned. If the input cell has an electron in its quantum-dot in the corner closest to the output cell, the cause is obvious. This example can be seen in Figure 2.6. Alternatively, if the input cell is oppositely polarised and there is no electron occupying this same quantum-dot, the cause is not directly apparent.

The physics of this situation can be compared to how two magnet bars interact when they are put parallel to each other with their poles aligned and one them is fixed in position while the other can only rotate. Technically the magnets are stable when they are perfectly parallel to each other, though as soon as this balance is lost only slightly, the mobile magnet will rotate itself perpendicularly to the immobile one. This loss of balance is imminent due to entropy, hence the configuration in which the two magnet bars are aligned perpendicular to each other is considered the most stable state, i.e. the least energetic configuration.

The 3MAJ-gate

The second essential gate, the three-majority-gate or 3MAJ-gate, occurs less frequently in conventional systems that allow for logical gates, however this three-input gate is very powerful in QCA designs because of its simplicity in implementation. Essentially, four cells are needed: three inputs cells and an output cell. The three input cells are positioned directly adjacent to the output cell, each on a different side. The output cell will conform, as the name suggests, to the majority of the three inputs. When two of the inputs have polarisation $P = +1$ and one input has polarisation $P = -1$, the output, which is affected equally by each input, will stabilise in polarisation $P = +1$ since the cumulative electrostatic energy that influences it to be polarised as such is greater than that of the opposite polarisation. While this opens up entirely new doors in the design of logical circuits compared to what is possible with conventional CMOS logic, note that the AND- and OR-gate can be created using this gate by fixing one of the inputs to polarisation $P = -1$ and $P = +1$ respectively as seen in Figure 2.8.

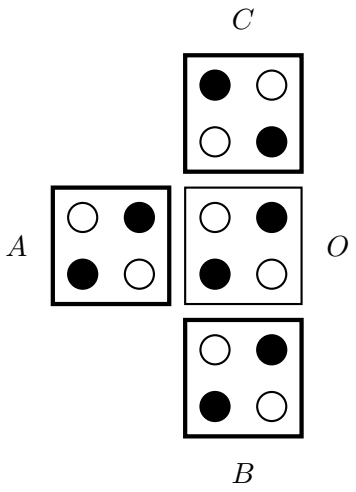
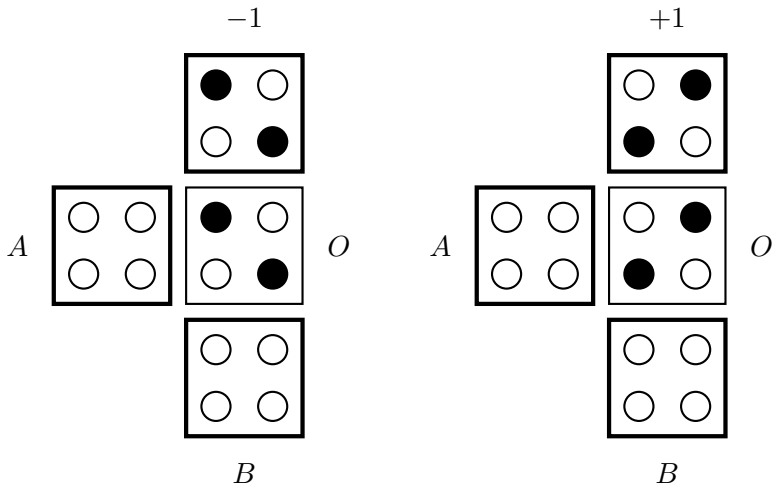


Figure 2.7: A 3MAJ-gate with inputs A, B and C and output O.
This example shows the majority of the inputs set to polarisation $P = +1$, causing the output cell to be most stable in the same polarisation.



(a) The 3MAJ-AND-gate. (b) The 3MAJ-OR-gate.

Figure 2.8: The AND- and OR-gate as 3MAJ-gates. Both have inputs A and B and output O. These logical gates include a fixed input cell polarised at -1 and $+1$ for the AND and OR-gate respectively. For the left (AND) gate, the output O will only change to $P = +1$ if both of the inputs become positive. Therefore: A AND $B \rightarrow O$. For the right (OR) gate, only one of the inputs polarised at $P = +1$ is needed to keep the output positive. If both inputs are set to $P = -1$, the majority of the inputs will be negative and the output cell will follow. A OR $B \rightarrow O$.

2.2 Bistable Approximation simulation

Since QCA designs are especially difficult to physically realise with the current state of technology, their functioning and stability is generally analysed with the use of digital simulation engines. Just like with CMOS designs, QCAs are usually designed with specialised software. In particular, the tool that is most commonly used for this is QCADesigner. This tool has been around for nearly two decades; it provides building tools for the design of QCAs and two simulation engines. The first one, the Coherence Vector simulation engine, is time-dependent and a less straightforward approach than the other. Time-dependent in this sense means that the simulation engine implements a function whose output is determined by time. We will focus on the second and most frequently used of the two, the Bistable Approximation simulation engine.

This simulation engine is time-independent and operates on the assumption that each cell can assume one of two states, similar to how the stability of a QCA cell was earlier defined as a two-state system with polarisations $P = -1$ and $P = +1$. The engine computes the state for each cell based on the states of neighbouring cells in a certain radius.

2.2.1 Kink Energy

At the core of both simulation engines found in QCADesigner is the Kink Energy function. It acts as a scaling factor for the mutual influence between two cells which decreases with distance. The function quantifies this energetic property for cells i and j by computing the electrostatic energies between every quantum-dot in i with each one j. The result is summed, first for when the cells have opposite polarisation, then the same is computed for when the cells have the same polarisation. The resulting kink energy between cell i and j ($E_{i,j}^k$) is produced by subtracting these two.

Formally, the function computing the electrostatic energy between quantum-dot i and quantum-dot j is defined as below.[1]

$$E_{i,j} = \frac{1}{4\pi\epsilon_0\epsilon_r} \frac{q_i q_j}{|r_i - r_j|}$$

Here, r_i denotes the location vector of quantum-dot i in three-dimensional space, hence $|r_i - r_j|$ corresponds to the absolute distance between i and j.

A couple of constants are found in the equation. We will quickly go through most of them although we will not explore why they are there. In short, they are present to make the function simulate the physical process that is happening using real-world constants. In the next chapter, we will discuss the redundancy of these constants when purely looking at the logical functioning of the system.

- ϵ_0 and ϵ_r are permittivity constants. ϵ_0 is the permittivity in free space and ϵ_r is the relative permittivity. These constants represent how easily an electron can travel through free space and through the cell wall material respectively. The latter medium requires a lot more energy from an electron to travel through, hence ϵ_r is a high number relative to ϵ_0. QCADesigner's default value for ϵ_r differs from the other permittivity constant by 13 orders of magnitude.

- q_i is the charge energy of a quantum-dot i, directly proportional to the electron volt constant ($eV = 1.602 \times 10^{-19}$ J). It can be read as $P_i \times eV$ where $P_i = \pm 1$ depending on whether there is an electron occupying this quantum-dot or not. This leads to two different outcomes for $q_i q_j$: either eV^2 when the respective cells of quantum-dots i and j have the same polarisation, or $-eV^2$ if their polarisation is opposite.

[1] While researching this simulation engine by looking at its documentation and source code, an inconsistency between the two was discovered: assuming the kink energy equation in the documentation is correct, the source code erroneously divides by 4 an extra time. It strictly follows the equation in the documentation, multiplying $\frac{1}{4\pi\epsilon_0\epsilon_r}$ by $q_i q_j$ then dividing the result by $|r_i - r_j|$, though instead of putting $\pm eV^2$ for $q_i q_j$, they put $\pm\frac{eV^2}{4}$. In *global_consts.h* in the source code, they define the constant `QCHARGE_SQUAR_OVER_FOUR` that is used for this. QCADesigner's author was contacted on this but no response was received as of yet.

2.2.2 Calculating the polarisation

Now, with the function that computes the kink energy, all the elements required to compute the polarisation of a cell are present. The core of this equation consists of the summation of the polarisations of all neighbouring cells scaled with their respective kink energies relative to the cell for which the polarisation is to be computed. The complete equation to calculate the polarisation of cell i is formally defined below.[2]

$$P_i = \frac{\sum_j \frac{1}{2\gamma_j} E^k_{i,j} P_j}{\sqrt{1 + (\sum_j \frac{1}{2\gamma_j} E^k_{i,j} P_j)^2}}$$

Here:

- P_j is the polarisation of neighbouring cell j
- γ_j is the tunnelling energy of the electrons in cell j. This is inversely proportional to the barrier energy, also known as the clock. In the Hold phase, when the clock is high, the barrier energy is high and the tunnelling energy is low, causing $\frac{1}{2\gamma_j}$ to be high compared to when j is in the Relax phase and the clock is low. This leads to the clock being directly proportional to the effect the polarisation of j has on the polarisation of i.

Again – similarly with the constants from the previous equation – the clock, which has its lower and upper bounds as constants, is redundant as an arithmetic scaling factor. This will be discussed in the next chapter.

2.2.3 Simulating a QCA

The above equation enables the computation of the polarisation of a single cell in an environment of cells for a single moment. For bistable approximation simulation, this computation is repeated for every cell in the environment to complete a single iteration. This process iterates until for each cell in the environment, the difference between the polarisations of the cell calculated in iteration i and iteration $i-1$ is less than or equal to a user-set convergence tolerance (ε). At this point it is said that the automaton has converged to a stable state for this instant. This exit condition can be logically written as $\forall x \in Env : |P^i_x - P^{i-1}_x| \leq \varepsilon$, where Env denotes the relevant cell environment and P^i_x the computed polarisation of cell x in iteration i.

A QCA can be fully simulated by repeating this process for each consecutive moment starting at time 0 until a finish condition is met.

[2]Note that while cells in the Hold phase are always fully polarised to ± 1, the same does not go for cells that undergo a shift in barrier energy. The equation above allows polarisations to be evaluated to a decimal number, which makes sense for the latter case: if a cell in the Switch phase is slightly polarised to $P > 0$ by neighbouring cells, it will propagate this signal further, though not as strongly as a cell in the Hold phase.

Chapter 3

A functional QCA system: QCA-STACK

As a proposed alternative to CMOS, QCAs are typically considered together with the dynamics of their physical implementation. This approach to conducting QCA research enables products to act as blueprints to put into use when QCA fabrication technology reaches a certain readiness, although the physical aspects introduce an abundance of variables to consider and thusly hinder lateral exploration of the emergent functional system.

QCA technology is one paradigm in an expanding set of realisable Field-Coupled Nanocomputing (FCN) technologies that are similar on a conceptual level. Chances are that FCN will be the next big thing in digital information processing after CMOS. [29] Still there is no guarantee that the generic QCA system as presented in the previous chapter (the Metal-Island implementation: [27]) will be the victor in this; experimental results even favour other implementations. [8] By abstractifying from the conventional physical context of QCAs to a certain extent, we hope to contribute not only to the theoretical field of QCA technology directly, but concurrently to ambient or related fields of study.

* * *

In this chapter, our main focus will be the abstractification of the QCA system from factors that bind it to the physical implementation, forming a discretely simulating simulation engine: one clock cycle is simulated in four time steps as there are four clock phases, opposing QCADesigner's continuous simulation of the adiabatic clock signal switching.

QCADesigner has no support for the procedural extensibility of QCA designs: the ultimate subject of this thesis. Because of this, the choice was made to create a bespoke QCA simulation and design tool: **QCA-STACK**. With the theoretical QCA system being focal to this thesis, the reduction to its logical core became a key focus in the design of our simulation engine.

3.1 Bistable Approximation simulation adapted and made discrete

3.1.1 Simplifying the Kink Energy function

Consider the function to compute the electrostatic energy between two quantum-dots i and j from the previous chapter once again:

$$E_{i,j} = \frac{1}{4\pi\epsilon_0\epsilon_r}\frac{q_i q_j}{|r_i - r_j|}. \tag{3.1}$$

As annotated in Section 2.2.1, $q_i q_j$ can be read as $\pm eV^2$. We can write this as $p_{i,j} \cdot eV^2$ where $p_{i,j} \in \{-1, 1\}$. If $p_{i,j} = 1$, it signifies that the respective cells of i and j have the same polarisation. Accordingly, $p_{i,j} = -1$ indicates that these cells have opposite polarisation.

In any established context, i.e. a context where ϵ_r is defined, the only terms in Equation 3.1 that vary with differing i and j are r_i, r_j and $p_{i,j}$. We can thus write:

$$\exists c \in \mathbb{R} : E_{i,j} = c \cdot E'_{i,j}, \text{ with} \tag{3.2}$$

$$E'_{i,j} = \frac{p_{i,j}}{|r_i - r_j|}. \tag{3.3}$$

Since our objective is to create a simulation engine free from factors that bind it to the physical implementation of QCAs, we can abstract away from this constant c for our functional system and focus on defining $E'_{i,j}$. This diminishes the problem to two sub-problems for any i and j: calculating the absolute distance between i and j, and determining the value of $p_{i,j}$.

Calculating $|r_i - r_j|$

The position of a quantum-dot in a cell can be derived by taking the offset from the centre of its encapsulating cell. For cell i with location vector $V_i = (x_i, y_i, z_i)$, we can encode the positions of its quantum-dots in a 2×2 matrix over vectors in the following way, distributing them evenly in the enclosed space:

$$Q_i = \begin{bmatrix} \begin{pmatrix} x_i - \frac{L}{4} \\ y_i + \frac{L}{4} \\ z_i \end{pmatrix} & \begin{pmatrix} x_i + \frac{L}{4} \\ y_i + \frac{L}{4} \\ z_i \end{pmatrix} \\ \begin{pmatrix} x_i - \frac{L}{4} \\ y_i - \frac{L}{4} \\ z_i \end{pmatrix} & \begin{pmatrix} x_i + \frac{L}{4} \\ y_i - \frac{L}{4} \\ z_i \end{pmatrix} \end{bmatrix} = \begin{bmatrix} q_{i,11} & q_{i,21} \\ q_{i,12} & q_{i,22} \end{bmatrix}. \tag{3.4}$$

Remember that cells are equilaterals; L denotes the length of a side.[1]

[1] QCADesigner sets this value to 1 nanometre by default, though 18 nanometers is most commonly used. [25, 5, 13, 17]

Computing $|r_i - r_j|$ is now a trivial task; the Euclidean distance between the two location vectors of the respective quantum-dots i and j is taken. More concretely: let i be the quantum-dot in the upper left corner of some cell a and let j be the quantum-dot in the lower right corner of some cell b. We can calculate the distance $|r_i - r_j|$ by taking the Euclidean distance between the vectors $q_{a,11}$ and $q_{b,22}$ as follows:

$$|r_i - r_j| = d(q_{a,11}, q_{b,22}) = \sqrt{\sum_{k=1}^{3}(q_{a,11,k} - q_{b,22,k})^2}. \tag{3.5}$$

Defining function p and computing the kink energy

In the context of computing the kink energy between two cells, determining the value of $p_{i,j}$ for any i and j is a more convoluted task than computing the absolute distance between two quantum-dots. Since the actual polarisations of the cells are left unregarded, it is impossible to determine $p_{i,j}$ directly.

Like mentioned in Section 2.2.1, the kink energy between two cells is computed by subtracting the energy between them given the cells have the same polarisation from the energy between the cells when their polarisation is opposite. Using the definition of $E'_{i,j}$ from Equation 3.3 to determine the energy between two quantum-dots, we can formally put this as:

$$E^k_{i,j} = E^{k,diff}_{i,j} - E^{k,same}_{i,j}, \text{ with} \tag{3.6}$$

$$E^{k,diff}_{i,j} = \sum_{kl \in \{11,21,12,22\}} \sum_{mn \in \{11,21,12,22\}} \frac{p^{diff}_{kl,mn}}{|q_{i,kl} - q_{i,mn}|}, \text{ and} \tag{3.7}$$

$$E^{k,same}_{i,j} = \sum_{kl \in \{11,21,12,22\}} \sum_{mn \in \{11,21,12,22\}} \frac{p^{same}_{kl,mn}}{|q_{i,kl} - q_{i,mn}|}. \tag{3.8}$$

Here, kl and mn represent the quantum-dots in cells i and j respectively by assuming the values of indices of a 2×2 matrix.

To complete this definition, we need the functions p^{diff} and p^{same}. For the convenience of the implementation in Haskell, the approach of constructing these functions as nested 2×2 matrices of 2×2 matrices was favoured. These matrices, in the form of $A_{kl,mn}$, consist of $p_{i,j}$ values where $A_{kl,mn} = -1$ if the quantum-dots at $q_{i,kl}$ and $q_{j,mn}$ share one electron between them and $A_{kl,mn} = 1$ otherwise. Using $\divideontimes$ as the infix operator for this function, we can write this as:

$$A_{kl,mn} = q_{i,kl} \divideontimes q_{j,mn}. \tag{3.9}$$

The following matrix emerges when, for each quantum-dot of cell i, $\divideontimes$ is applied to that quantum-dot with each one of cell j.

$$\begin{bmatrix}\begin{bmatrix} q_{i,11} \divideontimes q_{j,11} & q_{i,11} \divideontimes q_{j,12} \\ q_{i,11} \divideontimes q_{j,21} & q_{i,11} \divideontimes q_{j,22}\end{bmatrix} & \begin{bmatrix} q_{i,12} \divideontimes q_{j,11} & q_{i,12} \divideontimes q_{j,12} \\ q_{i,12} \divideontimes q_{j,21} & q_{i,12} \divideontimes q_{j,22}\end{bmatrix} \\ \begin{bmatrix} q_{i,21} \divideontimes q_{j,11} & q_{i,21} \divideontimes q_{j,12} \\ q_{i,21} \divideontimes q_{j,21} & q_{i,21} \divideontimes q_{j,22}\end{bmatrix} & \begin{bmatrix} q_{i,22} \divideontimes q_{j,11} & q_{i,22} \divideontimes q_{j,12} \\ q_{i,22} \divideontimes q_{j,21} & q_{i,22} \divideontimes q_{j,22}\end{bmatrix}\end{bmatrix}. \tag{3.10}$$

The matrices p^{diff} and p^{same} for two cells respectively having opposite and same polarisation can be derived from the above matrix:

$$p^{diff} = \begin{bmatrix}\begin{bmatrix} -1 & 1 \\ -1 & 1\end{bmatrix} & \begin{bmatrix} 1 & -1 \\ 1 & -1\end{bmatrix} \\ \begin{bmatrix} -1 & 1 \\ -1 & 1\end{bmatrix} & \begin{bmatrix} 1 & -1 \\ 1 & -1\end{bmatrix}\end{bmatrix}, \text{ and} \tag{3.11}$$

$$p^{same} = \begin{bmatrix}\begin{bmatrix} 1 & -1 \\ 1 & -1\end{bmatrix} & \begin{bmatrix} -1 & 1 \\ -1 & 1\end{bmatrix} \\ \begin{bmatrix} 1 & -1 \\ 1 & -1\end{bmatrix} & \begin{bmatrix} -1 & 1 \\ -1 & 1\end{bmatrix}\end{bmatrix}. \tag{3.12}$$

Looking at these matrices, it is directly apparent that:

$$\forall kl\ mn \in \{11, 21, 12, 22\} : p^{diff}_{kl,mn} = -p^{same}_{kl,mn}. \tag{3.13}$$

This means we can rewrite Equation 3.8 and consequently Equation 3.6 to:

$$E^{k,same}_{i,j} = \sum_{kl \in \{11,21,12,22\}} \sum_{mn \in \{11,21,12,22\}} \frac{-p^{diff}_{kl,mn}}{|q_{i,kl} - q_{i,mn}|} \tag{3.14}$$

$$= -E^{k,diff}_{i,j}, \text{ and} \tag{3.15}$$

$$E^{k}_{i,j} = 2E^{k,diff}_{i,j}. \tag{3.16}$$

The constant (2) in the last equation can be merged with the constant c from Equation 3.2 and can thus be disregarded since our definition of the kink energy function abstracts away from constants. This produces:

$$E^{k}_{i,j} = E^{k,diff}_{i,j} = \sum_{kl \in \{11,12,21,22\}} \sum_{mn \in \{11,12,21,22\}} \frac{p^{diff}_{kl,mn}}{|q_{i,kl} - q_{i,mn}|}. \tag{3.17}$$

Implementing the kink energy function in Haskell

We can generate the values of p^{diff} with arithmetic on the column numbers:

$$p^{diff}_{kl,mn} = (-2l + 3) \cdot (2n - 3). \tag{3.18}$$

This is used in the definition of `calcKinkEnergy` , the function that computes the kink energy between two cells and returns it as a `Double`:

```
calcKinkEnergy :: Cell -> Cell -> Double
calcKinkEnergy c1 c2 = sum . toList . flatten .
    mapPos ( \(_,n) a ->
      mapPos ( \(_,q) b ->
        fromIntegral ( ( -2 * l + 3 ) * ( 2 * n - 3 ) )
        / absQDotDistance a b
      ) $ getQDots c2
    ) $ getQDots c1
```

In this definition:

- `Cell` is the object that represents a cell. It is uniquely identified by its three-dimensional location vector and has a number of additional fields that we will expound on in Section 3.1.2. For now, it is important to note that there exists a function `getQDots` that takes a cell as argument and yields a matrix of location vectors of its quantum-dots similar to the one from Equation 3.4.
- `absQDotDistance` computes the absolute distance between two quantum-dots like described in the section on calculating $|r_i - r_j|$.
- `sum . toList . flatten` returns the sum of values in a nested matrix.
- `mapPos` maps over the values in a matrix along with their respective row and column numbers. It is a part of Haskell's `Matrix` library

3.1.2 Computing the polarisation of a cell

Continuing the bottom-up approach of simulating a QCA, the next step is to compute the polarisation of a cell in an environment using our definition of the kink energy function. Preliminarily, however, we need to discuss properties of the QCA cell and the simulation state in order to grasp every aspect of the resulting polarisation function definition. In the following paragraphs, formal definitions are mainly given in Haskell code.

Properties of the QCA cell

The most obvious properties of a QCA cell are its location and current polarisation. The former is unique for each cell in an environment and can therefore be used to identify a cell, hence also to test the equality of two cells. In the Haskell implementation, this location is defined to have the type `Pos`, which is equivalent to a 3-tuple of `Double`-type numbers. A value of this type can be read as a (x, y, z) location vector.

```
type Pos = (Double,Double,Double)
```

The following code snippet shows properties of the QCA cell:

```
data Cell = Cell { loc      :: Pos
                 , pol      :: Double
                 , phase    :: Time -> Phase
                 , label    :: String
                 , isInput  :: Bool
                 , isOutput :: Bool }
```

Here:

- `loc` is the location vector of the cell in three-dimensional space.
- `pol` is the current polarisation of the cell as a `Double`.
- `phase` is the clock function. It takes discrete time values as an integer as argument and outputs the current clock phase the cell is in. We will discuss the intricacies of the discrete clocking system in the next paragraph.
- `label` is a textual label used to identify a cell in an output log.
- `isInput` and `isOutput` hold information on whether the cell is an input or output cell respectively. These fields are used in several functions as we will see later in this chapter. In short, input cells act differently than regular cells since they are externally controlled, while output cells are just regular cells whose polarisation is read out and appended to the output log at the end of each time step.

The continuous clock signal made discrete

Before we dive into our adjustments to the existing system, let us first consider QCADesigner's method of implementing the clock signal.

As discussed in Section 2.2.2, the tunnelling energy – or inversely proportional to it, the barrier energy – scales the effect a cell has on its neighbouring cells just as the kink energy does. Physically, this barrier energy moves with the clock as a continuous repeating signal. QCADesigner simulates this by dividing the total simulation in a user-set amount of samples, then running iterations on each consecutive sample. In particular, the value of the clock signal is taken at each sample and used to determine the polarisation of each cell in the environment at the time of the sample. The amount of samples recommended by the documentation comes down to 2000 samples per repetition of the clock signal.

With the creation of our system, we endeavoured to reduce the existing system to its logical core. This process includes the transformation of the continuous clock signal to a discrete one. As a first step, we define the range of time units as a discrete group with the following type synonym definition:

```
type Time = Integer
```

At each time, each cell can be in one of the clock phases that we know: Hold, Release, Relax or Switch. We define it in Haskell thus:

```
data Phase = Hold | Release | Relax | Switch
```

The `phase` function of the `Cell` type is now fully defined: each cell has its current phase as a function of discrete time units as a property. Since each phase has its own characteristics, we form a set of rules instead of employing the clock signal as a scaling factor. Take the following rule for example: if some cell `c` is in the Hold phase at time $T = \texttt{t}$, we want its polarisation to be the sign of its current polarisation, otherwise we do not alter cell `c`. We can implement this rule as a function that transforms a `Cell`:

```
setHoldPolarisation :: Time -> Cell -> Cell
setHoldPolarisation t c = if phase c t /= Hold then c
                          else c { pol = signum $ pol c }
```

A small note on Haskell record syntax: a type can be defined with records, like the definition of `Cell` on the previous page. Each record is simply a function that takes its parent type – in this case a `Cell` – as its first argument and returns a value or function of the defined type. Records can also be overwritten as seen in the code snippet above.

In Section 3.1.3, we will discuss how this adaptation to the system plays out in practise and what adjustments are needed in the complete simulation engine in order to stay true to the functioning of the existing one.

The simulation state

When simulating a QCA, there are a number of variables that the engine needs to keep track of. To achieve this, we will use Haskell's `State` monad and construct the simulation state object `SimState` that keeps track of our variables.

Most trivially, the engine needs to keep track of the cells in the current environment. We can name this variable the `cellEnv` and define it to be a list of `Cell`-types with the type `[Cell]`. Another important variable that the engine needs to keep track of is the time. It is, for instance, used in the `phase` function seen in the previous two paragraphs to determine the current phase a cell is in. We can simply define this as the field `time` with the type `Time`.

We can compile the aforementioned state variables, plus a few additional ones that we will need later, into records of the `SimState` type:[2]

```
data SimState = SS { cellEnv   :: [Cell]
                   , time      :: Time
                   , stability :: Stability
                   , inputs    :: [Input]
                   , inputBuf  :: Input
                   , outputs   :: Output }
```

Aside from the previously defined fields, the following are (partially) defined:

- `stability` keeps track of stability of a simulation state. We will go further into the use of this field in Section 3.2.1. The type `Stability` is a type alias of the type `Bool`:

  ```
  type Stability = Bool
  ```

- `inputs` and `inputBuf` are used to give inputs to the QCA. They will be further discussed in Section 3.2.3.

- `outputs` logs the outputs of the QCA. We will use and discuss this in Section 3.2.2.

Functions returning a `State SimState`-type can apply operations on the `State`'s current `SimState` object, such as getting to retrieve the current state of the cell environment, or setting to change the value of the `stability` flag, for instance. Such functions are consecutively applied to form a chain of operations.

[2]Two records have been left out from this definition since they are not directly relevant to the plain simulation of a QCA. The two records are `kinkCache` which caches the kink energy between each pair of cells in the environment for optimisation purposes, and `finishLog`, which keeps track of variables used to determine whether the QCA has finished. We briefly touch on the latter in Section 3.3.1.

Implementing the polarisation function in Haskell

Now that we defined the QCA cell and the simulation state and their respective properties, we can continue the engine definition by defining the polarisation function.

Recall the equation to compute the polarisation of a cell from Section 2.2.2:

$$P_i = \frac{\sum_j \frac{1}{2\gamma_j} E^k_{i,j} P_j}{\sqrt{1 + (\sum_j \frac{1}{2\gamma_j} E^k_{i,j} P_j)^2}}. \tag{3.19}$$

The term $\frac{1}{2\gamma_j}$ scales the polarisations of neighbouring cells with the tunnelling energy of the respective cells. Our simulation engine applies the clock signal in a different way as discussed in this section, hence this term can be omitted in the equation that we will use to implement the function that computes the polarisation of a cell. We define this equation as follows:

$$P'_i = \frac{\sum_j E^k_{i,j} P'_j}{\sqrt{1 + (\sum_j E^k_{i,j} P'_j)^2}}. \tag{3.20}$$

The function will take a cell as argument – the cell to compute the new polarisation of – and returns a `Double`. Since this result depends on the state of the simulation, we need to encapsulate this return type in the `State` monad and put our simulation state object `SimState` into effect. We thus define the function that computes the new polarisation of a cell:

```
newPol :: Cell -> State SimState Double
newPol c = ( \x -> x / sqrt ( 1 + x ** 2 ) ) . sum . (
      ( zipWith (*) . map pol ) <*> map ( calcKinkEnergy c )
    ) <$> getNeighbours c
```

All functions in this definition are either native to Haskell or have been defined previously, apart from the function `getNeighbours`. This function, typed `Cell -> State SimState [Cell]`, takes a cell as argument and returns all 'neighbouring' cells in the current cell environment. In particular, it filters out each cell from the current cell environment for which the absolute distance from the centre of that cell to the centre of the cell given as argument to the function is more than a user-set effect radius. The filtered cell environment is returned as output.

Naturally, the rationale for not considering the entire cell environment instead, each time the polarisation of a cell is calculated, is purely computational efficiency. The kink energy between two cells decays rapidly with increased distance between them, hence cells that are spaced further apart have a negligible amount of influence on each other as the kink energy between those cells approaches zero. Their interaction can thus be disregarded in a simulation.

3.1.3 Complications involved in the discrete simulation

The discrete simulation of QCAs offers a great advantage in terms of computational complexity. Iterations need not be run for each sample – around 500 times per clock phase – but instead for each clock phase. While this approach produces a logical system that accurately simulates QCAs in the same way QCADesigner does for most inputs, there is specifically one edge case where the QCA utilises qualities of the adiabatic clock switching process. We will discuss this anomaly by exemplifying the 3XOR-gate.

Additionally, as the continual sampling of QCADesigner's simulation engine makes multiple time steps for each time step of the discrete method of simulation, the former inherently matches the functioning of the physical implementation more accurately. To compensate for this shortcoming, QCA-STACK introduces a defined evaluation order to match the physical process that transpires. This will be discussed in the last paragraph of this section.

The 3XOR-gate

The 3-input XOR-gate – also referred to as the TIEO- (three input exclusive or) or 3XOR-gate; we use the latter here – as the name suggests, takes three inputs A, B and C, and outputs $A \oplus B \oplus C$. The gate was first presented in [3] and proves to be a useful asset for QCA full adder miniaturisation as it functionally consists of merely twelve cells. This will be further discussed in Chapter 4. An exemplary state of the gate is shown in Figure 3.1 below.

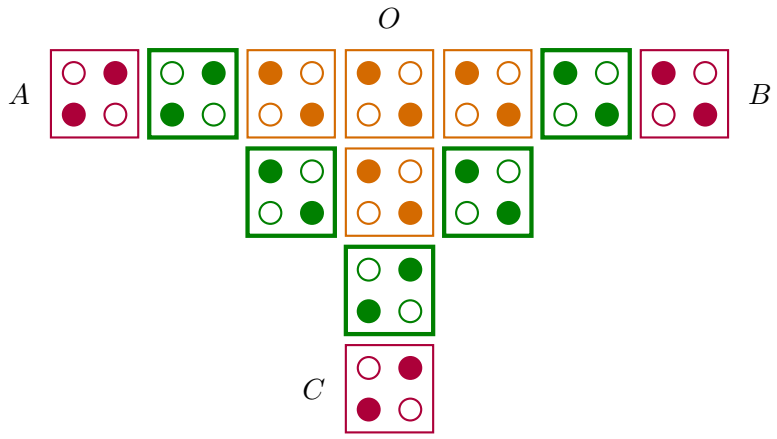


Figure 3.1: The 3XOR-gate with inputs A, B and C, and output O. The green cells are in the Hold phase and are therefore marked as driver cells. Also recall that differently coloured cells indicate different clock phases: orange follows green, green follows purple: the Release phase. The example displays the following polarisations: $A = C = +1$, $B = O = -1$. When converting the polarisations to logic 0 and 1, $A \oplus B \oplus C = O$.

While the example from Figure 3.1 does not illustrate the previously mentioned edge case, the logic behind the gate's behaviour for this input might not be directly obvious. Specifically, it is important to grasp how the cells in the second clock phase of the gate are polarised.

The cells directly adjacent to an input cell – a cell in the first clock phase of the gate – naturally assume the polarisation of their direct neighbour. The cells directly adjacent to inputs A and C both negatively polarise the cell diagonally between them, hence this cell is polarised to $P = -1$. The cell diagonally between the cells directly adjacent to inputs B and C, however, receives an equal amount of oppositely charged electro-static energy from its diagonal neighbours. Resultantly, the two energies cancel out and the smaller amount of negatively charged energy from its left non-adjacent neighbour thusly acts as the deciding factor.

A problematic case for the discrete simulation engine

This edge case arises when, contrary to the state shown in Figure 3.1, not one but both of the cells diagonally adjacent to cells directly adjacent to an input cell receive equal amounts of oppositely charged electro-static energy from both sides. This is the case for $A = 1$, $B = 1$, $C = 0$ and $A = 0$, $B = 0$, $C = 1$. The former case is shown in Figure 3.2 below.

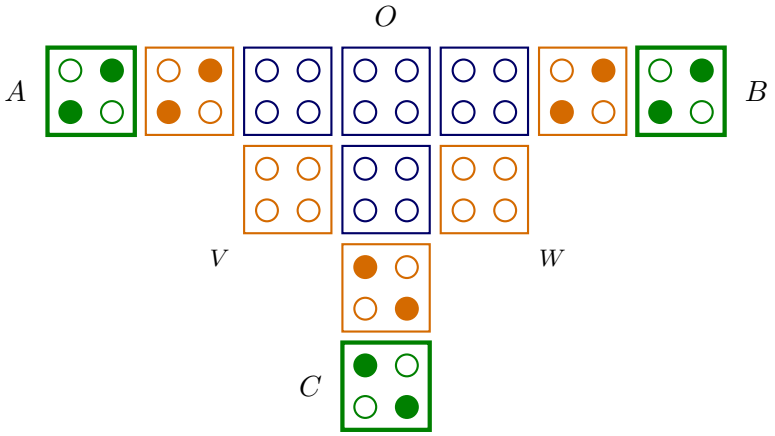


Figure 3.2: A 3XOR-gate again with inputs A, B and C, and output O. Inputs A and B are polarised to $P = +1$ whereas C is polarised to $P = -1$. Cells V and W cannot assume a clear polarisation in this state.

The above figure presents a difficult case for the discrete simulation engine as not all cells in the Switch phase (the cells depicted in orange) can assume a clear polarisation. In particular, the cells labelled V and W receive equal amounts of oppositely charged electro-static energy from either side of one diagonal. As discussed for the previous example state of this gate, the energies cancel out and a weaker interactant acts as the deciding force.

A naive discrete simulation engine would polarise V and W to the polarisation of inputs A and B, $P = +1$ in this case. Although the polarisations of inputs A and C cancel out for cell V, the faraway input B and its left neighbour will cause the cell to be polarised positively ever so slightly. Cell W is similarly polarised to $P = +1$ with the same reasoning.

This result is not consistent with the functioning of a 3XOR-gate, however. The 'sub-'QCA consistent of solely the orange and blue cells in Figure 3.2 is functionally equivalent to the 5-majority-gate (5MAJ-gate) presented in [18], meaning output O will polarise to the majority of the orange cells in the figure. Hence for the presented case, the positively polarised cells V and W cause output O to be polarised to $P = +1$ in the next phase. With two of the three inputs polarised to $P = +1$, a 3XOR-gate should output the polarisation $P = -1$. Continuous simulation of the QCA demonstrates that this gate functions appropriately anyway.

Continuous simulation using QCADesigner

There is a slight difference in the way QCADesigner implements the clock signal. Instead of the engine running one clock with each cell following this clock starting from their defined starting phase, QCADesigner produces four clock signals that run one phase apart. Each cell implements one of these four signals as the variable tunnelling energy of the electrons in the cell. A high clock in QCADesigner's output hence means high tunnelling energy and a therefore undefined polarisation, characteristic of the Relax phase. A low clock produces a defined polarisation, a quality of the Hold phase.

The 3XOR-gate was implemented in QCADesigner and its Bistable Approximation simulation engine was run with default settings on the input as depicted in Figure 3.2: $A = B = +1$, $C = -1$. Results gained from running the simulation engine are displayed in Figure 3.3. Three additional cells have been labelled: U is the cell between V and W, S and T represent the cells directly above – as seen in Figure 3.2 – cells V and W respectively.

The displayed simulation output shows only a fraction of the full output; other parts are not as relevant to understand how the polarisations of V and W are decided. Specifically, the Switch phase transition of the clock 2 – the clock implemented by the cells in the Relax phase in Figure 3.2 – is shown. At this point, clock 1 is constant low, meaning that V and W already went through the Switch phase and are now in the Hold phase. It can be seen, however, that their polarisation is not clearly defined and rests around the zero point, waiting for a deciding factor. The two cells are very slightly negatively polarised due to the negative effect cell U has on them. Cell U, even though the electrons in this cell have a high tunnelling energy, affects its left and right neighbours ever so slightly with the negative polarisation it receives from the cell below it, as well as the cells directly adjacent to the respective inputs cells B and C.

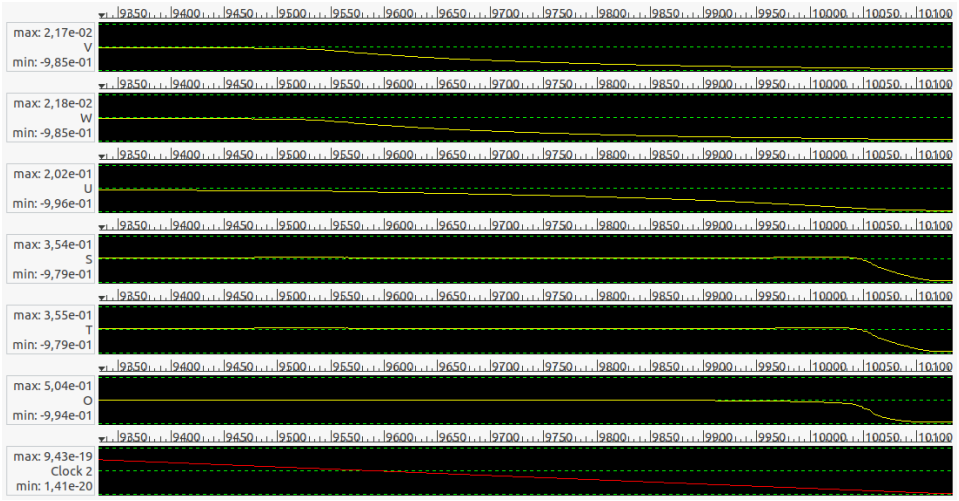


Figure 3.3: Output from QCADesigner's Bistable Approximation simulation engine, displaying the Switch phase transition of the cells previously in the Relax phase in the state depicted in Figure 3.2. The value of clock 2 and the polarisations of cells V, W, U, S, T and O are graphed over time. The numbers on the horizontal axes represent the sample numbers. The polarisation of a cell in the Relax phase is close to the middle green dashed line, as its polarisation not well defined.

The process of the 3XOR-gate for this case is hinges on a nominal deterministic race condition. Cells V and W, being in the Hold phase, will polarise quickly when given a predominant push in either direction, hence their polarisation depends which deciding push is given first. In Figure 3.3, it can be seen that cell U is the first to be affected by the decreasing clock signal, around sample number 9425.

The symmetry in the design allows us to reduce the comparison in energy received from neighbouring cells in the Hold phase for cell U and cells S and T. For instance, the symmetry makes the energy for cell S and cell T equivalent, hence we need only consider one of the two. Let the cells directly adjacent to inputs A and B be A_1 and B_1 respectively. The energies for cell U and cell S can be compared by calculating the following difference:[3]

$$(-1 \cdot E^k_{U,A_1}) - (+1 \cdot E^k_{S,B_1}) \approx 0.0025. \tag{3.21}$$

While the difference is not significant, it seems to be enough for cell U to win the race condition. Cells V and W quickly assume the negative polarisation, therewith further polarising cell U. Cell S balances around the zero point due to the energies accumulated from cells A_1, V and U until finally switching when cell O starts polarising negatively.

[3]The calcKinkEnergy function from Section 3.1.1 was used to compute this.

Solving the edge case for the discrete method

The race condition that decides how certain cases involving one or more cells with an undefined polarisation in the Hold phase resolve, remains difficult to translate directly to a discrete system. Although this problem is certainly not impossible to solve, its complexity increases with increasingly complicated edge cases as a result of the energetic interactions that take place on the three-dimensional plane, continually changing over minute time frames.

A solution for a simulation system that could perhaps be proven to be equivalent to the existing continuous simulation system, is an engine that acts as a hybrid discrete-continuous one. It would act as a discrete simulation system, but instead of solving an edge case much like the one discussed by analysing the convoluted network of energies at one state, it would simulate this phase transition continually instead. While there could be more certainty on the accuracy of a semi-discrete simulation engine, simplicity was favoured for the design of the fully discrete simulation engine presented in this thesis.

Motivated by the perceived rarity of this edge case, being – so far – only encountered in the context of the 3XOR- and the 5MAJ-gate, a discrete solution that works especially well in the encountered contexts was opted for. The method emanated from an alternate deduction of how the polarisation of cells V and W is decided for the case discussed in the preceding two paragraphs.

Consider how cells V and W polarise when these cells are left out of the equation when computing the polarisations of the cells in the succeeding phase, and we instead compute the polarisations of cells V and W when those polarisations are decided. We can model this scenario by modifying the clock phases of these cells to be two phases later, as shown in Figure 3.4 below. The subsequent phase in shown in Figure 3.5.

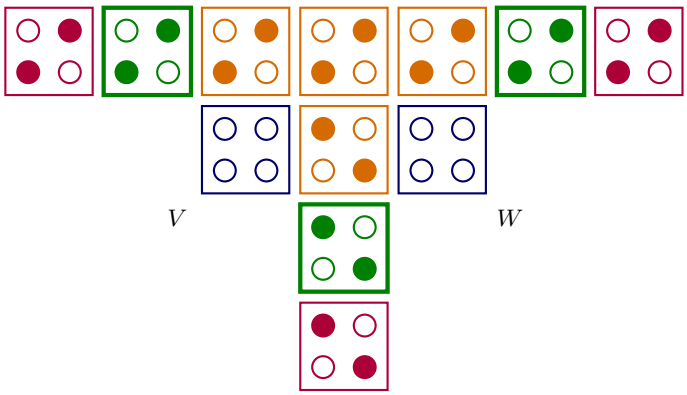


Figure 3.4: The 3XOR-gate, modified to illustrate our scenario.
As cells V and W have moved from the Hold phase to the Relax phase, we say that their respective clocks have been incremented by two.

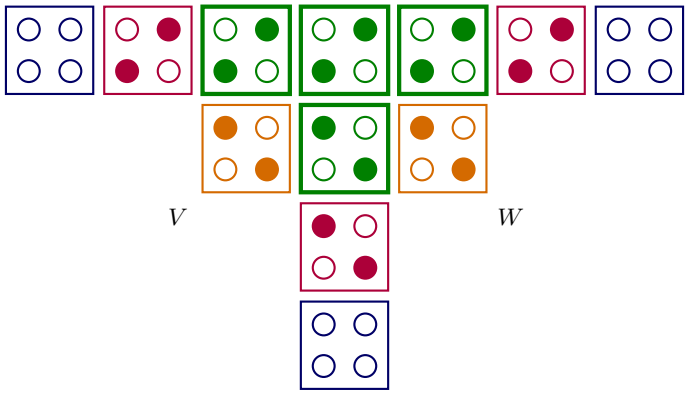


Figure 3.5: The modified 3XOR-gate displaying the state following the state shown in Figure 3.4 by one phase transition.
Cells V and W assume the polarisation of the majority of their directly adjacent neighbours in the Hold phase. Both thusly polarise to $P = -1$.

The above figure shows that cells V and W acquire the polarisation $P = -1$. This is the correct polarisation for the edge case we discussed, hence we could use the method that obtained these results on dubious cells in general. Naturally, this approach also produces a correct result for the other edge case in which the input polarisations are inverted, as this causes each polarisation to be inverted.

For the last step of our method, we roll the simulation back to the state where the edge case arose and plug in the acquired polarisations for the problematic cells. For the case that was extensively discussed, this results in the state shown in Figure 3.6 below.

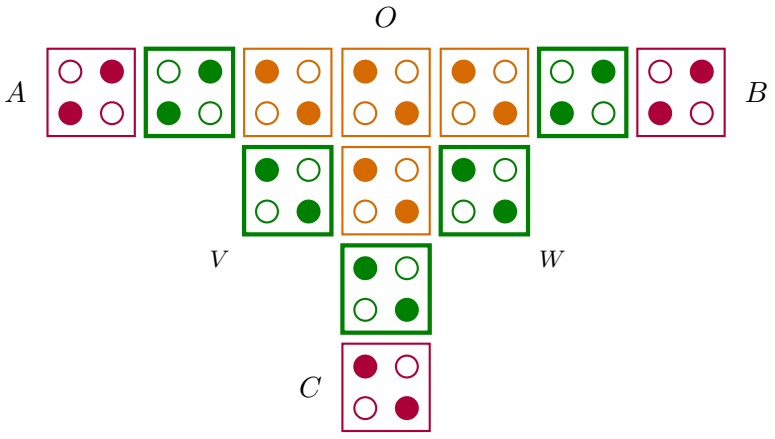


Figure 3.6: The 3XOR-gate, now with the correct polarisations plugged in for cells V and W. This leads to output O correctly polarising to $P = -1$.
In logical terms: $A = B = 1$, $C = 0$, $A \oplus B \oplus C = 0 = O$.

As discussed previously, we make no claims on the accuracy of this method for the general case of dubious cells and no further research was done on this due to time constraints. Evaluating the accuracy of our method or finding an alternative provably correct method is left as a interesting topic of research for future works.

The importance of evaluation order with discrete simulation

During the development of the discrete simulation engine, one other issue was observed; the simulation of certain designs produced unexpected results that were inconsistent with the existing simulation solution. This occurred due to the naïve approach to cell evaluation order that was implemented.

When simulating cells in an environment – as will be discussed in detail in Section 3.2.1 – the cells are evaluated in a certain order. As polarisations of cells are sequentially computed, cells that are evaluated later in the sequence consider the newly computed polarisations of cells that were evaluated earlier in the sequence. When simulating discretely, certain environments can develop differently depending on the order of evaluation as will be shown with the upcoming example.

QCADesigner's simulation engine randomises the evaluation order by default, though the evaluation order matters much less for this method of simulation. Simulating by continually sampling a continuous signal matches the reality of the adiabatic switching process much closer, hence the small-step approach, by definition of the adiabatic theorem (discussed in section 2.1.2), produces states of global minimum energy with better accuracy. For the discrete simulation engine, a different method is required to match the states that eventuate from the physical processes that occur during a phase transition. We present our method in the next paragraph.

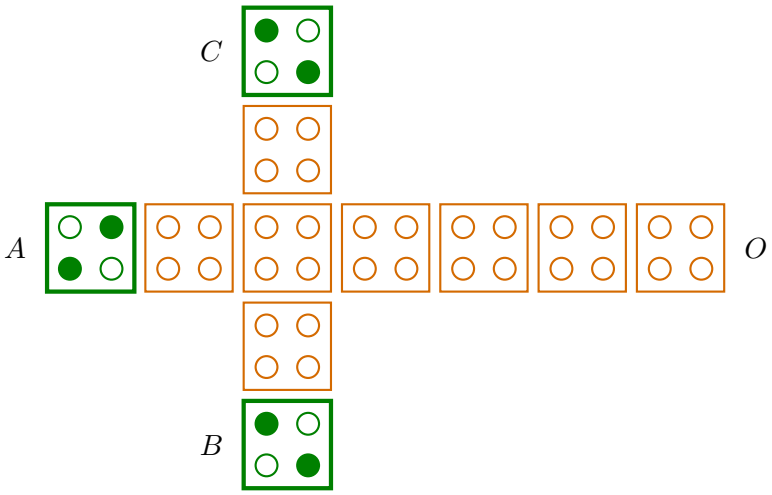
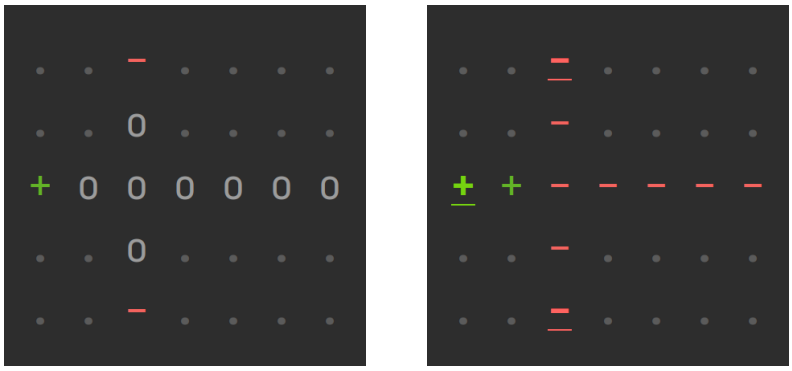


Figure 3.7: An atypical 3MAJ-gate with inputs A, B and C and output O.

Consider the 3MAJ-gate shown in Figure 3.7. This shape, in its essence, occurs in [23] and will be used in Chapter 4 of this thesis. The important difference with the 3MAJ-gate discussed in Section 2.1.3 is that the cell on position $(x, y) = (x_B, y_A)$ does not differ in clock phase with its left and vertical adjacent neighbours. Nevertheless this gate functions like a 3MAJ-gate should, at least when simulated correctly.

We can use QCA-STACK's visual output function that will be discussed in Section 3.3.2 to display the gate more compactly. Figure 3.8 below depicts the gate for some input, together with the correct subsequent state.



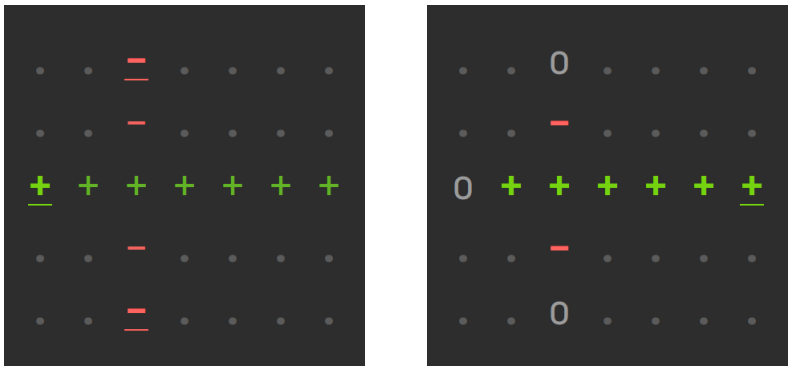
(a) QCA-STACK's render of Figure 3.7.

(b) The correctly simulated subsequent state.

Figure 3.8: The 3MAJ-gate from Figure 3.7 in two consecutive states, rendered and simulated using QCA-STACK with its visual output function. Characters + and − represent polarised cells in the Switch phase. Cells in the Hold phase are represented either by their bold-faced versions, or by bold and underlined characters in the case of input or output cells.

We define a naming convention for cells in Figure 3.8 based on their location on the grid: cell C is on grid position $(x, y) = (2, 0)$ and hence named c_{20}; cell B is named c_{24}. Now consider what happens when polarisations of the cells are evaluated in the following order: first the cells adjacent to an input cell are evaluated in any order, then sequentially cells c_{32}, c_{42} and c_{52}, and finally cells c_{22} and c_{62} in any order. The first set of cells will clearly polarise to their nearest input cell. The next cell in line, cell c_{32}, receives negated energy from its two diagonal neighbours, thus polarising to $P = +1$ and propagating this signal further to cells c_{42} and c_{52}. At the intersection, cell c_{22} receives more positive energy from the horizontal axis than negative energy from the vertical one, hence it polarises to $P = +1$ as well. Output cell O assumes the polarisation of its left neighbour and thereby produces an output that is inconsistent with the expected functioning of the 3MAJ-gate.

Two consecutive states of the simulation of the 3MAJ-gate with the aforementioned order of evaluation are depicted in Figure 3.9.



(a) The erroneously simulated state.

(b) The state following the one shown in Figure 3.9a.

Figure 3.9: The 3MAJ-gate from Figure 3.7 in two consecutive states, simulated with QCA-STACK using a specific order of cell evaluation. The sub-figures show how output O is incorrectly polarised to $P = -1$ when simulated with a naïve approach to cell evaluation order, yet the majority of the inputs is polarised to $P = -1$.

Sorting on propagation order

When considering the physical processes that take place when simulating the 3MAJ-gate with the inputs from the example in the previous paragraph, it seems only logical that cell c_{22} should be evaluated before cell c_{32}. As the cell barriers are being lowered, the cells previously in the Switch phase gradually polarise. The cells closer to a driver cell polarise more quickly as they receive more polarising energy. They will propagate this newly acquired polarisation further, though only slightly since they are not fully polarised and the barriers are not fully lowered, hence only affecting very nearby cells. This property of polarisation propagation through adjacent cells can be compared to the functioning of a wire, in which a signal reaches nearer areas sooner than ones further away. Following this property when choosing the cell evaluation order ensures cell c_{22} is evaluated before cell c_{32}.

To implement this for the QCA-STACK engine, we can design a function that sorts the cell environment on the rule of propagation, before sequentially evaluating the cells in the environment. We define this rule in natural language as follows: a cell that is closer to a driver cell should be evaluated before a cell that is further away from one. More formally, this is:

$$a < b \iff MD(a, Env, t) < MD(b, Env, t), \tag{3.22}$$

where $MD(a, Env, t)$ returns the absolute distance from a to the nearest driver cell in Env at time $T = t$. This result can be obtained by filtering Env for cells that are in the Hold phase at that time and returning the absolute distance from a to the nearest one.

3.2 Simulating a QCA

We continue the definition of QCA-STACK's discrete simulation engine. In the previous section, we defined the core function `newPol`, and discussed methods to improve the simulation accuracy in the absence of multi-step simulation of the adiabatic switching process. With these tools, we are now able to construct the core of the discrete simulation engine.

3.2.1 Simulating one time step

We start by defining the elements needed to simulate one time step. This begins with describing how one iteration is run on a cell environment.

Simulation rules

As briefly described in Section 2.2.3, one iteration of the simulation is completed by sequentially evaluating the polarisation of each cell in the environment. The process repeats until, for each cell, the difference between the respective polarisations set in the current and the previous iteration is not more than the user-set convergence tolerance. Defining this in Haskell starts by defining how the polarisation of a cell is determined. This, among other things, consists of computing the new polarisation with function `newPol`, defined in Section 3.1.2. As stated in that section, the discrete simulation engine takes a rule-based approach to simulation, instead of simulating a function with samples of a continuous clock value. We hence need to define the rules that need to be applied to determine the polarisation of a cell.

We informally define the rules to determine the polarisation of a cell as follows. We will later define it formally in Haskell. For any cell c:

- **If c is an input cell, we do not change its polarisation.** Supplying polarisations to input cells is handled separately by the simulation engine, this will be handled in detail in Section 3.2.3.
- **If c is in the Switch phase, we compute the new polarisation.**
- **If c is in the Hold phase, we do not change its polarisation.** There is one exception to this. We discussed in Section 3.1.3 that cells can have an undefined polarisation in the Hold phase. If this is the case, we want to evaluate the polarisation of the cell anyway.
- **If c is in the Release phase, we set its polarisation to zero.**
- **If c is in the Relax phase, we do not change its polarisation.** The polarisation of a cell in the Relax phase should always be zero. Since regular cells are always initialised with no polarisation and since the previous rule states that polarisations are set to zero in the preceding phase, there is no need to change anything in this phase.

There are two additional rules that apply when computing the new polarisation for a cell.

- If the difference between the current polarisation and the newly computed one is more than the convergence tolerance, we set the `SimState`'s `stability` flag to `False`, which is set to `True` before the iteration is run. By this rule, the new polarisation of each cell in the environment must converge in order to produce a stable simulation at one time step, i.e. keep the `stability` flag set to `True` after the polarisation of each cell in the environment has been evaluated.
- If the polarisation of a cell in the Hold phase is evaluated, we set this cell's polarisation to the sign of the newly computed polarisation. This operation is also part of function `setHoldPolarisation` that was defined as an example in Section 3.1.2.

Evaluating the polarisations of cells in an environment in Haskell

Cells are sequentially evaluated using the `tick` function. This function implements the rules described in the previous paragraph. We define it as:

```
tick :: Time -> Cell -> State SimState ()
tick t c
 | ( phase c t == Hold && pol c /= 0 )
  || phase c t == Relax || isInput = return ()
 |   phase c t == Release           = setPol c 0
 | otherwise = newPol c >>= \p ->
     when ( abs ( p - pol c ) > convergenceTolerance )
          ( setStability False ) >>
     setPol c ( if phase c t == Hold then signum p else p )
```

Two functions that occur in this definition are non-native to Haskell and have not been defined in earlier sections.

```
setPol       :: Cell -> Double -> State SimState ()
setStability :: Stability      -> State SimState ()
```

These setter functions update the `State` monad by modifying a field of the `SimState` object. In the case of function `setPol`, the `cellEnv` field is updated with a modified polarisation for one of its elements. Additionally, we define constant `convergenceTolerance` as a global constant along with a default value:

```
convergenceTolerance :: Double
convergenceTolerance = 1e-8
```

The `tick` function is called in sequence for each cell in the environment in function `tickAll`, defined below. One iteration has been run when this function finishes; it then returns the current value of the `stability` flag.

```
tickAll :: State SimState Stability
tickAll = get >>= \st ->
    foldr ( (>>) . tick ( time st ) ) ( gets stability )
    $ cellEnv st
```

A brief note on Haskell's `State` monad and how it is used in our context. Function `get` retrieves the current state of the `SimState` object. In all definitions in this thesis, such `SimState`-typed variables will be named `st`. Function `gets` takes one argument, a function that takes a `SimState`-type as its only argument (a `SimState` record for instance), applies this argument to the current state of the `SimState` object, and returns the result encapsulated in `State SimState`. Lastly, we will encounter function `modify`, which takes a state updating function as its only argument.

Simulating multiple iterations

As discussed earlier in this section and in Section 2.2.3, a time step is simulated by running multiple iterations until, for each cell in the environment, the newly computed polarisation converges. The QCA is said to be converged to a stable state, a state of minimal global energy. The lower the convergence tolerance is set, the longer it takes for each cell in the environment to converge, and the more accurate the simulation as there is more certainty about the minimality of the global energy in the resulting state.

Since the number of iterations that is run is a considerable scaling factor in the computational complexity, we allow the user to limit the number of iterations to a certain maximum. This means that the exit condition from Section 2.2.3 should be modified to the following:

$$i > \text{Max Iterations} \vee \forall x \in Env : |P_x^i - P_x^{i-1}| \leq \varepsilon. \tag{3.23}$$

We define function `doIterations`, which takes the maximum amount of iterations as argument. Each time before running an iteration with the `tickAll` function, the function sets the `stability` flag to `True` and sorts the cell environment with a call to function `sortOnPropagation`. The former is done so the function can verify whether the entire cell environment converged after running the iteration, as each non-converging polarisation computation changes this flag to `False`. The latter is necessary for simulation consistency due to reasons explained in the paragraph on evaluation order in Section 3.1.3. The last paragraph of the latter section describes the sorting operation that is being performed here.

When the function is done iterating, it calls function `getDubiousCells`, also defined below. This function filters the environment for cells in the Switch phase that are not polarised enough to consider their polarisation defined: so-called dubious cells or *dubiants*. Their case has been extensively discussed in Section 3.1.3. Returning this set of cells enables us to monadically chain the function `handleDubiants` directly after function `doIterations`; the set is passed to it as the final argument. The function applies the solution described in Section 3.1.3 to decide the polarisations of dubious cells and is defined below.

```
doIterations :: Integer -> State SimState [Cell]
doIterations 0 = getDubiousCells
doIterations n = setStability True >> sortOnPropagation >>
                 tickAll >>= \case
                     True  -> getDubiousCells
                     False -> doIterations $ n - 1
```

```
getDubiousCells :: State SimState [Cell]
getDubiousCells = gets $ \st ->
    filter ( \c -> abs ( pol c ) < 0.05
                && phase c ( time st ) == Switch )
    $ cellEnv st
```

```
handleDubiants :: Integer -> [Cell] -> State SimState ()
handleDubiants maxIters uds = unless ( null uds ) $ do
    ce <- gets cellEnv
    modify $ \st ->
      st { cellEnv = filter ( `notElem` uds ) ce }
    t <- nextPhase
    _ <- doIterations maxIters ; setHoldPols $ t + 1
    modify $ \st -> st { cellEnv = cellEnv st ++ uds }
    foldr ( (>>) . tick t . ( \c -> c { pol = 0 } ) )
          ( return () ) uds
    modify $ \st ->
      st { cellEnv = filter ( `elem` uds ) ( cellEnv st )
                  ++ filter ( `notElem` uds ) ce
         , time = time st - 1 }
```

The last definition uses the previously undefined functions `nextPhase` and `setHoldPols`. The former increments the `time` field, then executes the latter function. It will be further defined and used in the next section. Function `setHoldPols` fully polarises all cells in the Hold phase at the time given as argument, by effectively applying the function `setHoldPolarisation` that was defined in Section 3.1.2 to each cell in the environment.

3.2.2 Implementing the simulation function in Haskell

In the previous section, we defined the functions required to simulate a single time step with multiple iterations. In this section, we apply these methods to produce a function that simulates multiple time steps of a QCA and logs the output at each time step. We start by defining the latter process.

Logging simulation output

At each time step, after the iterations have been run, we want to log the simulation output to track the evolution of certain cells specified as output cell. This is done by executing function `addOutputs`, which filters the cell environment for output cells: cells that have their `isOutput` field set to `True`. The output contains a list of output cells in their current state along with the current values of the `time` and `stability` fields, and is prepended to the `outputs` list of the `SimState` object. This field is defined to be of type `Outputs` in Section 3.1.2, which is a type alias defined as follows:

```
type Output = [ ( (Time,Stability) , [Cell] ) ]
```

The function is defined as follows:

```
addOutputs :: State SimState ()
addOutputs = modify $ \st -> st { outputs =
    ( ( time st , stability st )
    , filter isOutput $ cellEnv st ) : outputs st }
```

The simulation function

We now define the core function `simulate`. The function runs recursively to simulate all time steps, incrementing the `time` field with every step, until a certain finish condition is met. In this definition, this finish condition is decided by function `isFinished`, which returns a `State SimState Bool`. We do not define this function here as it would make this section unnecessarily complex, though we will briefly touch on it in Section 3.3.1. Additionally, function `setInputCells` is used in the definition. This method does as its name suggests and is described in a separate section, Section 3.2.3.

```
simulate :: Integer -> State SimState ()
simulate maxIters = setInputCells >>
    doIterations maxIters >>= handleDubiants maxIters >>
    addOutputs >>
    isFinished >>=
    flip unless ( nextPhase >> simulate maxIters )
```

Function `nextPhase` was previously mentioned. It first calls function `incrTime`, which updates the state by incrementing the `time` field by one and returns the resulting value, then it calls function `setHoldPols`, described at the end of Section 3.2.1, and returns the earlier obtained return value, the current value of `time`.

```
nextPhase :: State SimState Time
nextPhase = incrTime >>= (<$) <*> setHoldPols
```

Users do not call the `simulate` function directly, instead they call the function defined in Section 3.2.4. This function sets up the initial state with the supplied cell environment and inputs and executes the simulation.

3.2.3 Asynchronously supplying input values to the engine

Designing an input system for a discrete QCA simulation system is no uncomplicated task. Since QCA-STACK is designed to be used as a command-line tool, the program should be able to handle user-given inputs that are not too complicated for the user to construct. Before we look at why this is a complicated task, we first consider how inputs should be given by the user.

Giving inputs to the engine

Consider a very simple QCA, consisting only of a 3MAJ-gate. The QCA has three inputs, labelled A, B and C. The user now wants to simulate a clock cycle with the following inputs: $A = +1$, $B = +1$, $C = -1$. This could be supplied to the engine in the following format:

```
type Input = [(Cell,Charge)]
```

Here, `Charge` is simply defined as:

```
data Charge = Negative | Neutral | Positive
```

Given that the `Cell`-types `a`, `b` and `c` respectively represent cells A, B and C, our user will thus supply:

```
inp :: Input
inp = [ (a,Positive) , (b,Positive) , (c,Negative) ]
```

Now consider that the user wants to simulate the gate with different inputs for each consecutive clock cycle. The input type becomes a list of `Input`-types, each consecutive element destined to be used in a consecutive clock cycle. Say the user wants to simulate first the previous input, then the input: $A = -1$, $B = +1$, $C = +1$. The user supplies:

```
inp :: [Input]
inp = [ [ (a,Positive) , (b,Positive) , (c,Negative) ]
      , [ (a,Negative) , (b,Positive) , (c,Positive) ] ]
```

If all input cells are in the same clock phase, supplying input values is trivial. The first item of the input list is taken each time the cells are in the Switch phase and the polarisations of the input cells are set to their corresponding input values (this involves the simple process of converting a `Charge` to a `Double`). This item is then removed from the input list so that the same operation can be performed next time the inputs are set. The problem increases in complexity when an `Input` cannot be set in a single clock phase, as the input cells are in different clock phases.

Per clock input buffering

When QCA designs scale up in size and complexity, the probability that all input cells are defined to start in the same phase becomes slim. Since we do not want to give the user the task of giving inputs for each consecutive phase, we need a method that accepts the input from the previous example, even if the input cells are out of phase. This can be done by using an input buffer for each clock. We describe the procedures as follows:

PREPARE Let an operation be applied to the user-given input that sorts the input into a list of four `Input`-types, one for each clock. Same-phase inputs are inserted in the same index of this list while maintaining their respective order; inputs that should be given earlier occur earlier in the `Input`. We now have four input buffers, one per clock.

LOAD Each time we set inputs with `setInputCells`, we sequentially load elements from the input buffer in which each cell is currently in the Switch phase – the current-clock input buffer (CCIB) – and put them in another buffer: the active input buffer (AIB). The AIB will only accept inputs of which the input cell is unique in the buffer. After this operation has been applied to each input in the CCIB, we remove from it the cells that were added to the AIB.

SET After each loading operation, we apply one of the following operations on inputs in the AIB depending on their current phase:

Switch We set the input cell to the corresponding input value.

Release We set the polarisation of the input cell to zero. The input is removed from the AIB.

Other We do nothing.

The **PREPARE** procedure is performed once before the simulation starts; each call to function `setInputCells` invokes procedures **LOAD** and **SET**.

Formatting user-given inputs to per clock buffers

In essence, the `[Input] -> [Input]` transformation from a list of sequential user-given inputs to a list of input buffers is no more complicated than the application of the three following operations: we `concat` the lists to an `Input`-type with all inputs, we `sort` these inputs on the clock of the respective input cells, and finally we `group` the inputs on equal clocks. This produces an `[Input]`-type of list size 4. Every input with same-phase input cells ends up in the same list, while maintaining the original order between the inputs within each list.

The problem becomes more complicated when we consider the following example of a larger scale, propagating QCA: let Q be a QCA with input cells I_n for $n \in N$ with $N = \{0 \dots 31\}$. We define I_0 to have clock 0, and each consecutive input is one phase apart, hence we say I_1 has clock 1 and so on. Resultantly, I_4 will have clock 4, which is equivalent to clock 0. These phased inputs allow signals to propagate through Q, so the signal that starts at I_0 is progressively influenced by subsequent input cells.

A naïve input system with per clock buffers would not only set input cell I_0 upon the first call to function `setInputCells`, but all input cells in the following set:

$$\{I_n \mid n \in N \wedge n \bmod 4 = 0\}.$$

This leads to erroneous signal propagation through Q. When the first input is applied to I_0, the signal propagates further, receiving input values from the first input from I_1 through I_3, before then receiving a input values from the second input from I_4 through I_7. I_8 through I_{11} apply input values from the third input and so on. We hence need to offset these inputs in the buffer so that the first signal will receive an input value from the first input for every I_n for $n \in N$.

We do this by prepending our user-given input with `Neutral`-value inputs for the input cells we want to offset, before we perform the **PREPARE** procedure. Since this procedure respects the order of inputs, the first **LOAD** procedure will load the AIB with `Neutral`-value inputs for input cells we want to offset, thereby offsetting the values from first input for these cells by one clock cycle. By recursively applying a method that finds cells we need to offset, we create layers of offset: two layers for input cells I_8 through I_{11}, three for I_{12} through I_{15} and so on.

We define function `offsetBuf` that recursively finds layers of input cells we need to offset. The function takes the ordered (I_0 before I_1) list of input cells of the QCA as argument and assumes that the first input will always begin in the Switch phase. We apply the function and prepend its output to the user-given input with the following function application:

```
( map (,Neutral) ( offsetBuf inCells ) : ) userInput
```

```
offsetBuf :: [Cell] -> [Cell]
offsetBuf []   = []
offsetBuf inps = (++) <*> offsetBuf $ offsetInputs inps
  where
    offsetInputs = dropWhile ( \c -> phase c 0 /= Switch )
                 . dropWhile ( \c -> phase c 0 == Switch )
```

The `State` fields `input` and `inputBuf` are the formatted per clock buffer and the AIB respectively.

3.2.4 Running the simulation engine

Thus far, we defined functions to be used by the engine itself to simulate a QCA, though none specifically designed for the user to invoke. In this section, we define function `runSimulation`: the most fundamental way to produce simulation output from user input. In Section 3.3.2 and Section 3.3.1, we will show other methods that the user can invoke for more user-friendly and insightful simulation output, or for more control on the running length of the simulation.

There are three arguments a user supplies to the simulation engine. First and foremost, the cell environment. This is simply the collection of cells that make up the QCA. Additionally, the user supplies the (unformatted) input that they desire to simulate, as well as the maximum amount of iterations to control the running time of the simulation. Our function executes function `execState` from Haskell's `State` library; this takes a `State`-returning function as its first argument that modifies the initial state given as second argument: the provided `SimState` in our case. It returns the resulting state.

We define the function thus:

```
runSimulation :: Integer -> [Input] -> [Cell] -> Output
runSimulation maxIters inps ce = outputs $
    execState ( simulate maxIters ) defaultState
      { cellEnv = ce
      , inputs  = parseInputs ( getInputs ce ) inps }
```

Here:

- `defaultState` is defined as the 'blank' state, with all list-type fields set to the empty list, `time` set to 0, et cetera.
- `parseInputs` performs the **PREPARE** procedure described in the previous section. It calls function `getInputs`, which filters the cell environment for cells with `isInput` set to `True` and sorts them on their (previously undefined) `number` field. This field is useful when upscaling a QCA from n inputs to $2n$ inputs as will be discussed in the next chapter.

3.3 Other noteworthy features of QCA-STACK

Previously in this chapter we discussed the core functionality of QCA-STACK: the discrete simulation of QCAs through user input. In the next chapter we will cover functionality that especially distinguishes the system from the conventional QCA designing tool QCADesigner: programmatical extensibility of QCA designs, the feature that mainly motivated the creation of QCA-STACK. There is more functionality that has been added to our system than we have previously covered or will cover in Chapter 4, however. This section highlights a selection.

3.3.1 Finish conditions

We previously encountered function `isFinished` in the `simulate` function presented in Section 3.2.2. This function decides whether the QCA has finished, by evaluating whether the last input has been propagated to the last output cell and output of this cell is recorded in the Hold phase.

This is an especially hard problem to solve for larger scale QCAs, as there is no uncomplicated approach to deciding by how many clock phases the last input and output cells differ *in practise*, instead of taking their absolute clock phase difference. Designs can be made that propagate a signal from the last input cell to the last output cell over multiple clock phases, hence it would require an algorithm that analyses the cell environment to consistently find the correct clock phase difference. Alternatively, this difference could be determined by the user and given as argument to the simulation engine.

Due to time and scope limitations, none of the aforementioned methods were implemented. Instead, a smaller scale automatic last-input-propagated algorithm was designed. We will not describe how it works in all of its intricacies, as the process is not straighforward and thus out of scope for this chapter. In short, it uses the `finishLog` field of the `State` object – mentioned in the footnote in Section 3.1.2 – to keep track of the highest-numbered output cell that recorded output in the Hold phase.

To give more detail, the algorithm starts keeping track when the input buffer of the first clock is empty, taking the number of the current highest-numbered output cell in the Hold phase as the highest number – this is stored in the `finishLog` field. With every call to the function it then looks for an output cell in the Hold phase with a number not lower than the current highest number, until an output cell is found with its number equal to the maximum number of output cells in the cell environment. In this case, function `isFinished` returns `True` to indicate that the QCA has finished.

Additionally, to give the user more control over the simulation length, an analogue to function `simulate` was formed that takes the maximum amount of phases as argument. It replaces the call to function `isFinished` with a test of the value of the `time` field (T) against this argument (T_{max}).

3.3.2 Visual and pretty-print outputs

In Section 3.2.4, we discussed function `runSimulation`, which produces raw output in the form of an `Output`-type. While this unprocessed type of output is useful for debugging purposes, it is not particularly easy to interpret. Figure 3.10 below shows such output.

```
λ> runSimulation 50 ( take 2 . truthTable $ getInputs threeMajCellEnv ) threeMajCellEnv
[((6,True),[
 3MAJ:     -1]),((5,True),[
 3MAJ:     -0.6330161496159146]),((4,True),[
 3MAJ:      0]),((3,True),[
 3MAJ:      0]),((2,True),[
 3MAJ:     -1]),((1,True),[
```

Figure 3.10: Simulation of a 3MAJ-gate with function `runSimulation`. The function `truthTable` is used here. This function generates a truth table from its input, to be used as user-given input to the simulation engine. Also note that list of outputs is reversed; earlier recorded outputs have higher indices. This is due to an optimisation, as prepending an element to a list in Haskell is of lower time complexity than appending it.

Pretty-print output

An algorithm was designed that groups the simulator output into sequential output groups corresponding to sequential inputs. The procedure of this algorithm will not be discussed since this is again out of scope for this chapter. The function is named and typed as follows:

```
prettyPrintIO :: [Input] -> Output -> IO ()
```

It is invoked by a `runSimulation` analogue: function `runSimPretty`. The latter function differs from the former in two places. Firstly, instead of taking raw input as its second argument as function `runSimulation` does, it takes a function that acts as a input generator. It can be seen in the type definition of function `runSimPretty`:

```
runSimPretty :: Integer -> ( [Cell] -> [Input] ) -> [Cell]
                        -> IO ()
```

The input generator function takes a list of input cells as argument, obtained by the application of function `getInputs` to the third argument supplied. This enables the user to supply the cell environment only once when running the simulation engine with an input generator, as opposed to passing it twice as seen in the example shown in Figure 3.10. Supplying direct inputs now requires the user to simply utilise Haskell's `const` function.

Secondly, the output – as well as the unprepared input – is applied to function `prettyPrintIO`. This produces prettily printed outputs where outputs are shown together with their respective inputs, as seen in Figure 3.11 below.

```
λ> runSimPretty 50 ( take 2 . drop 4 . truthTable ) threeMajCellEnv
 ============

 A:         1
 B:         0
 C:         0

 3MAJ:      0

 ============

 A:         1
 B:         0
 C:         1

 3MAJ:      1

 ============
```

Figure 3.11: Simulation of a 3MAJ-gate with function `runSimPretty`. The design has three inputs cells, labelled A, B and C. The output cell is labelled as $3MAJ$.

Visual output

The pretty-print output mode, while effective as a clear input-output view, does not give insight into inter-cellular interactions. The former mode – as generated by function `runSimulation` – does give detailed information on the polarisations of the output cells at each time, though being limited to output cells only, it does not grant insight into the dynamics of the entire QCA. Such insight can, however, be gained with a visual output mode.

We describe the requirements the visual output must meet to enable the user to debug a QCA by visually analysing the inter-cellular interactions. The output should:

- show the polarisation of each cell in the environment at each time step.
- show the relative positions of the cells in the environment.
- support multi-layered (three-dimensional) designs.

Single-layered (two-dimensional) designs are easily printed as the output medium (a computer screen) is two-dimensional itself, though multi-layered designs are harder to render. An uncomplicated solution was opted for: layers are printed side by side, with the lowest layer shown on the left side.

Visual output is generated by running the simulation engine with the function `runSimVisual`, typed equivalently to function `runSimPretty`. The function invokes another analogue of the `simulate` function that, instead of logging outputs with function `addOutputs`, appends a graphical representation of the current cell environment to an output string. The string is returned and printed by function `runSimVisual`. Output is printed to the console after each clock phase as a result of Haskell's lazy evaluation.

The graphical representation consists solely of ASCII characters, though infused with ANSI escape sequences to introduce colours and differing typefaces. In particular, cells currently in the Hold phase are printed as bold characters while otherwise printed as regularly-faced characters. Additionally, input and output cells in the Hold phase are rendered as bold and underlined characters. Cells in the Release or Relax phase are printed as `o`.

Examples of output generated by function `runSimVisual` are seen in Figures 3.8 and 3.9 of this chapter, as well as in figures found in Chapter 4. Figure 3.12 below exemplifies visual output of a multi-layered QCA.

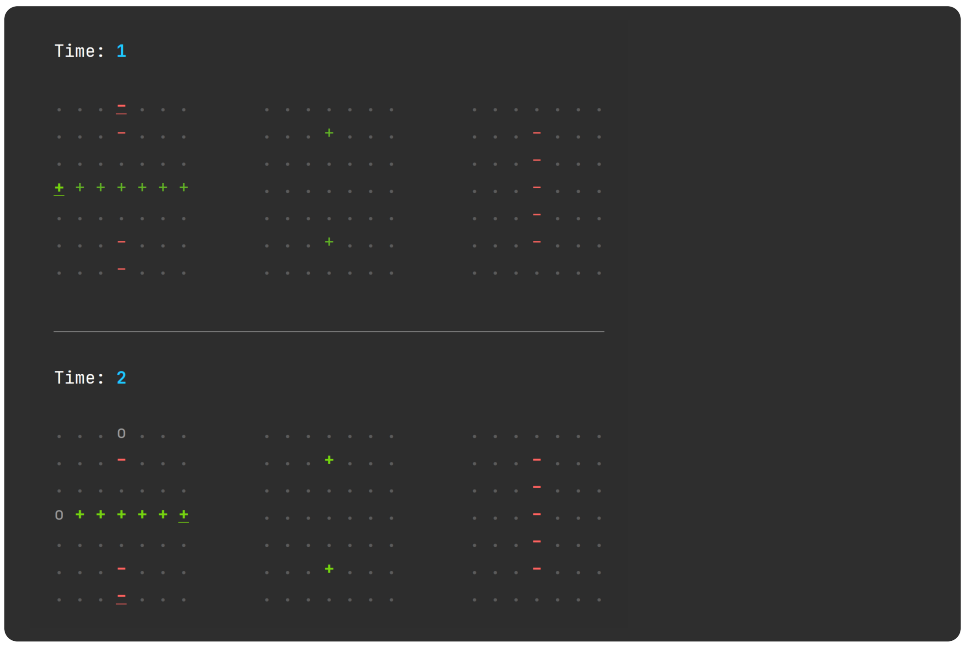


Figure 3.12: Example output generated by the visual output mode. The output shows two simulated time steps of a multi-layered wire crossing that utilises the third dimension to form a bridge. The underlined cells at time $T = 1$ are input cells, whereas those at time $T = 2$ are output cells.

[3] QCA-STACK was (for the most part) created in the Jetbrains IntelliJ IDEA. The visual output function was therefore designed for IntelliJ's built-in console. Other Windows CLIs such as Windows PowerShell are able to display ANSI-escaped text sequences also, though colours and spacing could differ from the intended output.

3.3.3 Input files

The simulation of a QCA requires the user to define the cell environment in Haskell as a `[Cell]`-type, which includes defining each individual cell in the list separately. Fortunately, QCA-STACK includes template cells so that defining most cells only requires an altered location and phase function in most cases. As an example, the cell environment `threeMajCellEnv` used in Figures 3.10 and 3.11 is defined in Figure 3.13 below.

```
λ> i1 = inputCell { label = "A"    , loc = ( -1 ,  0 , 0 ) }
   i2 = inputCell { label = "B"    , loc = (  0 , -1 , 0 ) }
   i3 = inputCell { label = "C"    , loc = (  0 ,  1 , 0 ) }
   o = outputCell { label = "3MAJ" , loc = (  0 ,  0 , 0 ) , phase = clock 1 }
   threeMajCellEnv = [i1,i2,i3,o]
```

Figure 3.13: Cell environment `threeMajCellEnv` defined in Haskell. Function `clock` is used in the definition; this generates a `phase` function with the starting phase depending on the argument given. Template cells are defined to start in the Switch phase with `clock 0`.

While this method of defining cell environments is convenient enough for small QCAs, the task becomes tedious when defining larger designs. As a solution to this, we present a feature that enables QCA-STACK to be used as a design tool: QCA definition input files.

A description of the input file format

The input files begin with a textual representation of the cell environment, we call this the layout definition. Layers are delimited with a row containing only `=` characters, one for each column. Each character in the layout definition is separated by exactly one space; trailing spaces are permitted on each line of the input file.

Each non-space character in a layer represents a cell in the environment. The value of their `loc` field is derived from the position in the layout definition. The characters can be one of the following:

- A `+` or `-` character represents a fixed positive or negative input cell respectively. These are found in the AND- and OR-gates shown in Figure 2.8.
- A number $n \in \{0..9\}$ represents a regular cell with its `phase` function set to `clock` (n mod 4).
- Any letter in the ranges `A-Z` and `a-z`. The other section of the input file uses these letters as identifiers in order to define fields for these cells. Equal letters on different positions therefore produce cells that differ in the `loc` field, though have the same other fields.

After the layout definition comes the other section of the input file: the cell definitions. The two sections are separated by the `$` character, directly preceded and followed by a minimum of one newline character. The cell definitions that follow are delimited by a minimum of one newline character and formatted thus: first the identifying letter directly followed by a colon, then, on the following subsequent lines in any order, the flags. The flag definitions begin with a `-` character followed by a space, after which one of the following flags is put:

- `input` or `output` respectively sets the `isInput` or `isOutput` flag to `True`.
- `clock` = n sets the `phase` function to `clock` (n mod 4).
- `label` = "a" sets the `label` field to a; any non-" character is accepted.
- `number` = n sets the `number` field to n; n should be a natural number.
- `offset` = (x,y,z) adjusts the `loc` field by adding the given x, y and z values. This is useful for defining cells with one or more non-integer coordinates, as coordinates inferred from the layout definition grid can only be integers.

Setting any of the aforementioned overwrites (or adjusts) the default value for the appropriate `Cell` field; not setting a flag is therefore also an option.

The content of an input file representing an AND-gate is shown below.

```
= = =
  -
a 0 o
  b
= = =

$

a:
- input
- label = "A"

b:
- input
- label = "B"

o:
- output
- label = "A /\ B"
- clock = 1
```

Chapter 4

Stackable QCAs

In the previous chapter, we discussed our bespoke QCA system and the intricacies of its method of simulation. The system was conceived not for the purpose of offering replacement for the conventional QCA simulation solution (QCADesigner), but instead to offer the possibility to programmatically alter cell environments with ease. In particular, the capability to create a generalised method to stack a QCA on top of itself was sought. In theory, the emergent system of the multi-layered QCA allows for unbounded three-dimensionality; we utilise this concept to form 2^n-extensible QCAs.

In this chapter, we will consider the multi-layered QCA conceptually and discuss its appearance in related works. We then present primary components of the ALU as 2^n-bit extensible designs: the ripple carry adder (RCA), the bitwise logical operations module (AND/OR specifically) and the multiplexer (MUX). The first two scale linearly and are thus comparatively easy to implement, whereas the multiplexer scales partly logarithmically. This introduces a non-trivial problem; we propose a solution in Section 4.4.2.

Throughout this chapter, we assume the reader is familiar with the functioning of the essential logic gates (Section 2.1.3) and the 3XOR-gate (Section 3.1.3). QCA-STACKs visual output mode will also make appearances; instructions on the interpretation of this output are found in Section 3.3.2.

The QCA-STACK input files associated with designs presented in this chapter are found in Appendix A. In particular, the designs presented in Sections 4.2.2, 4.3 and 4.4.1 can be found there. The same input files and input files of other designs are also found on the QCA-STACK GitHub page.

4.1 Extending QCAs in the third dimension

Before we consider the use of stacking QCA designs and methods to do so, let us first consider the practise of designing multi-layered QCAs along with its benefits and possible disadvantages, as well as the conventional use of multi-layering in quantum-dot cellular automata.

4.1.1 Multi-layered QCAs

There has currently been around twenty years of research on QCAs. Most works published in the field present a new design and compare it to similar existing designs. A good proportion of these are multi-layered designs, though almost each multi-layered design that is presented in a publication consists of strictly three layers. One publication was found that presents a 5-layer design; this appears to be the single exception to the pattern. [7] In practise, a three-layered QCA amounts to a dual-layer QCA since the middle layer is nearly always used as a tool to propagate a signal upwards.

Consider two cells stacked directly on top of each other, with the lower cell as the driver cell. Each electron in the lower cell pushes the electrons in the other cell away, hence the latter cell assumes a polarisation that is opposite of that of the former. When another cell is put above the two, the signal negation takes place twice and the integrity of the signal originating from the lowest cell is therefore kept; this is shown in Figure 4.1.

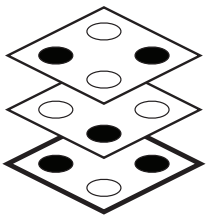


Figure 4.1: Three cells stacked on top of each other.
The signal originating from the lowest cell is conducted to the highest cell.

This concept is used to form conventional multi-layered QCAs, functionally consisting of two layers connected by a conducting middle layer. One of the most practical uses is for a wire crossing; this is shown in Figure 3.12. Furthermore, it is used for the design of ultra-dense processor components such as ripple carry adders; the two logical gates the design consists of are placed above each other and connected to inputs to form the adder.

The literature that was consulted makes no statement on why specifically three layers are used and not more. The apparent scarcity of publications presenting a $n > 3$ multi-layered design makes one wonder about the attainableness of a physical implementation of such a system. Since we focus on the theoretical side of the QCAs in this thesis in the field of computing science, we disregard possible concerns on the physical implementation of $n > 3$ multi-layered designs and assume the QCA system is fully functional in the third dimension. Additionally, for the ease of designing three-dimensional QCAs, we compute the layer separation needed to make the kink energy between a cell and its coplanar directly adjacent neighbour equivalent to the negated kink energy between the same cell and its vertical (non-coplanar) neighbour; this is done in the next subsection.

4.1.2 Pragmatical layer separation

Since we seek to utilise the third dimension as simply another dimension, we need the absolute kink energy to be equivalent between coplanar and non-coplanar cells of the same Euclidean distance to any cell. This is similar to how the second dimension is introduced to a one-dimensional QCA system: any two cells that are adjacent in a one-dimensional QCA have the same kink energy as any two cells that are directly adjacent in a two-dimensional QCA.

Layer separation is simply implemented by scaling all z values with a certain constant when computing a Euclidean distance. When the z-scalar (Z_s) is set to 1 so that there is no scaling, two non-coplanar neighbouring cells c and c_a, according to QCA-STACK, have a kink energy of $E^k_{c,c_a} \approx -0.11$. Two coplanar neighbouring cells c and c_b, however, have a kink energy of $E^k_{c,c_b} \approx 0.82$. We need to compute the value of Z_s such that $E^k_{c,c_a} \approx 0.82$. The approach to computing the value is as follows:

- We calculate the exact value of the kink energy between cells c and c_b, where cell c has location vector $V_c = (0, 0, 0)$ and cell c_b has $V_{c_b} = (1, 0, 0)$. Let this be E^k_{c,c_b}.
- We calculate the exact value of the kink energy between cells c and c_a, with $V_{c_a} = (0, 0, Z_s)$. Let this be E^k_{c,c_a}.
- We define the following equality: $E^k_{c,c_b} = -E^k_{c,c_a}$. We solve the equation for Z_s.

The resulting equation is the following:

$$-\frac{4}{Z_s} + \frac{8}{\sqrt{Z_s^2 + \frac{1}{4}}} - \frac{4}{\sqrt{Z_s^2 + \frac{1}{2}}} = -\left(\frac{4}{3} + \sqrt{\frac{64}{5}} - \sqrt{\frac{8}{5}} - \sqrt{8}\right) \tag{4.1}$$

Considering the difficulty of obtaining an exact value for Z_s and the gain of having an exact value instead of an approximation, it was opted for to enter the equation in the online calculator Wolfram|Alpha.[1] This calculator was able to approximate a numerical solution for Z_s with increasing accuracy each time the 'More digits' button was clicked. The result approximates to the following:

$$Z_s \approx 0.581215 \tag{4.2}$$

Supplying the simulation engine with a larger decimal number increments the simulation running time. By setting Z_s with six decimals, the two kink energies start differing after the fifth decimal. This was deemed sufficiently accurate.

[1] https://www.wolframalpha.com

4.2 Ripple Carry Adder

Arithmetic operations are a core feature of a processing core. Some of the primary components of the ALU that is responsible for these operations are the RCA, the bitwise logic module and the multiplexer; we discuss the first in this section.

There have been many publications in the field of QCAs that have presented new and/or improved designs of the RCA, either single-layer or multilayer. [23, 18, 16, 12] Here, we present stackable RCA, based on the design presented in [23]. Stackable in this sense means that we can apply an operation to the design to transform an n-bit RCA into a $2n$-bit RCA. The base RCA that we present is 2-bit and consist of two modified versions of the RCA presented in [23] stacked on top of each other where the latter is rotated and reflected in a way that enables the carry out to propagate directly into the carry in of the stack that is to be placed on top of it.

The RCA – with inputs A, B and C_{in} (carry in) and outputs SUM and C_{out} – functionally consists of two core components: a 3MAJ-gate and a 3XOR-gate. The inputs are connected to the gates to produce outputs in the following way:

$$3MAJ(A, B, C_{in}) = C_{out}, \text{ and} \tag{4.3}$$

$$3XOR(A, B, C_{in}) = SUM. \tag{4.4}$$

The corresponding truth table confirms that this holds:

A	B	C_{in}	C_{out} $3MAJ(A,B,C_{in})$	SUM $3XOR(A,B,C_{in})$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 4.1: The truth table of an RCA along with the outputs of the 3MAJ- and 3XOR-gates.

The two gates, very similarly to [23], are put above each other – the 3MAJ-gate on the bottom layer – with the input cells on the lower layer propagated upwards to serve as inputs to the 3XOR-gate as well. This constitutes a 1-bit RCA.

4.2.1 The 2^n-bit RCA

Multi-layered designs take up a lot of space to display two-dimensionally; we hence use the visual output mode to display the design. The 2-bit base design consists of nine layers: four layers for each 1-bit RCA (of which half are conducting layers), and one layer that contains only the upwards propagated C_{out} output signal. This last layer is merged with the first layer of the RCA that is stacked on top of it; the cell with the C_{out} signal takes the place of the new C_{in} input cell here. Since both gates constituting the 1-bit RCA – and particularly the 3MAJ gate that produces the C_{out} – have an input-output clock phase difference of one, the second 1-bit RCA is offset by one clock phase as it requires the C_{out} output from the first 1-bit RCA.

As example input to the RCA, we compute the sum $3+2$. This amounts to the following inputs: $A_0 = A_1 = B_1 = 1$, $B_0 = C_{in} = 0$ (A_0 and B_0 are the inputs of the lower 1-bit RCA). The simulation starts at time $T = 0$, where the first input cells are in the Switch phase. We show the first four layers of the 2-bit RCA at times $T \in \{1, 2\}$ in Figure 4.2; the second set of four layers is shown at times $T \in \{2, 3\}$ in Figure 4.3. The ninth layer is not shown; it contains a single cell directly above the cell in the layer below it.

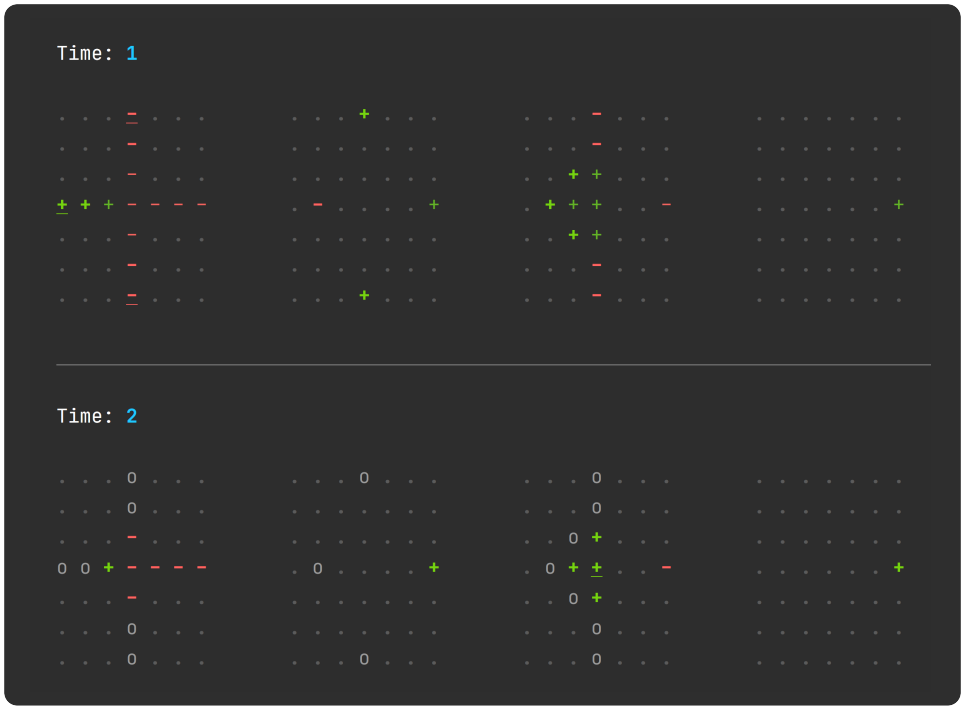


Figure 4.2: The lower 1-bit RCA with inputs C_{in} (top), A_0 (left) and B_0. The first layer contains a 3MAJ-gate that produces the C_{out} output, which is propagated upwards. The third layer contains the SUM_0 output cell.

Figure 4.3: The upper 1-bit RCA with inputs A_1 (left) and B_1 (bottom). The C_{out} signal from the lower 1-bit RCA is propagated upwards and taken as the third input to the 3MAJ-gate (right). Output cell C_{out} is polarised to the output of the 3MAJ-gate ($P = +1$); output cell SUM_1 is polarised in the third layer shown ($P = -1$). Together with the polarisation of output cell SUM_0 ($P = +1$), the outputs represent the number 5.

4.2.2 The 2^n-bit RCA: Version 2

The previously shown design of the 2^n-bit RCA has an issue when it comes to using the design as a components of a larger design: the SUM output signals are difficult to propagate further, as their respective obvious exit paths are blocked by the upward-propagating C_{out} signals.

We present a solution to this that does not include an additional clock phase. Propagating the C_{out} signal to the edge of the design within the same clock phase introduces complications, as this outward-propagating path is prone to influence from other areas of the design. The resulting design functions correctly when simulated with a higher number of maximum iterations, though seems not entirely stable for a lower number. The design could therefore be considered not robust.

The simulation output of the earlier discussed version of the 2^n-bit RCA shown in Figure 4.3 is shown in Figure 4.4 as the second version.

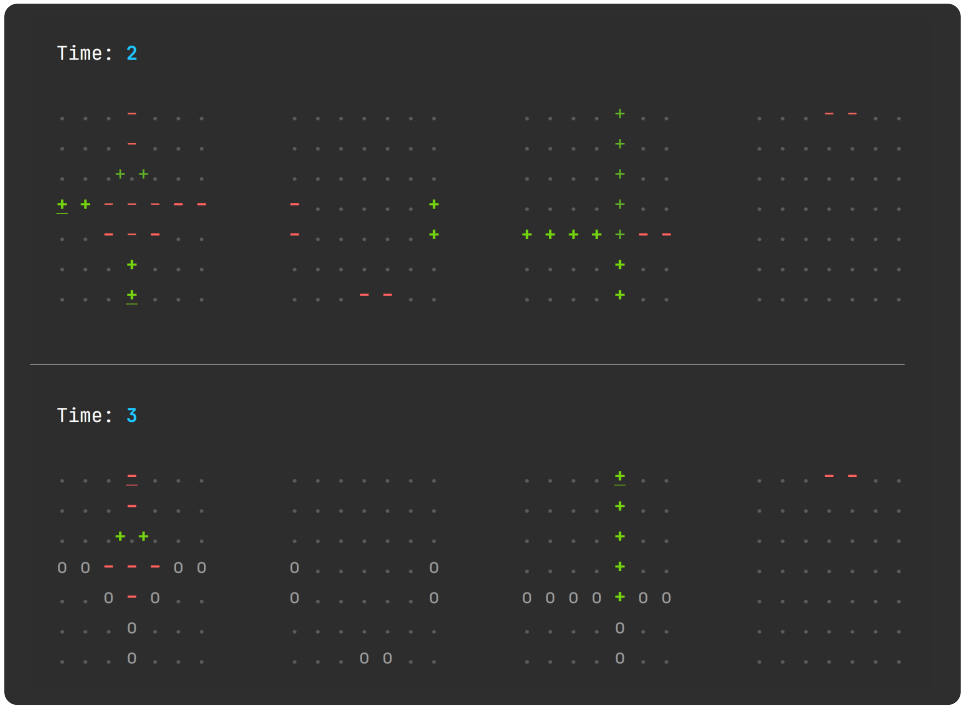


Figure 4.4: The upper 1-bit RCA of the second version of the 2^n-bit RCA. There is no change in the input cells, though the 3XOR- and 3MAJ-gate layers have been swapped. Additionally, the output of the 3MAJ-gate is shifted to the right such that the interference between that output and the output of the 3XOR-gate is minimised.
In the layer shown top-left, it can be seen that two cells have non-integer x coordinates; this was set using the `offset` flag of the input file, discussed in Section 3.3.3. The visual output mode is able to display cells at x coordinates ± 0.5, though y and z coordinates must be integers.

4.2.3 A general linear stacking solution

In this paragraph, we present the method to apply to, for instance, any 2^n-bit RCA to turn it into a 2^{n+1}-bit RCA. The method is designed in such a way that it can be applied as a general solution for linearly stackable designs; we show another example in Section 4.3.

Our method needs to distinguish two different types of input and output cells. In particular, there are regular input and output cells such as A_0, B_1 and SUM_1, and there are propagation input and output cells like C_{in} and C_{out}. To define this distinction, we extend the definition of the `Cell` type with the field `propagate` of type `Bool`. This field can be set in an input file by adding the flag `propagate` to a cell, just like how the flags `input` or `output` are added.

The function, named `stackDesign`, takes an argument of type `Int` (n), another argument of type `(Int,Int)` ($t = (a, b)$), and the input cell environment of type `[Cell]`. It basically duplicates the cell environment and returns the result, though some alterations are done to the original cell environment and the duplicated one. The original cell environment gets just one adjustment: output cells with the `propagate` field set to `True` – output cell C_{out} for instance – are converted to normal cells. That is, the `output` and `propagate` fields are set to `False`.

The duplicated cell environment (the new stack) undergoes more modifications and uses the parameters n and t. This first argument facilitates the process of determining the size of the input stack, which is, for instance, used to number the new cells. Function `stackDesign` is called with $n = 2^0$ when stacking a design for the first time, then with $n = 2^1$ when stacking the result again and so on. The second argument parametrises properties of the base design.

The following modifications are applied to the new stack:

- Input cells with the `propagate` field set to `True` are removed. This removes input cell C_{in} from the new stack, for example.
- Each cell is translated upwards: the z coordinate is incremented by the maximum z coordinate value in the original cell environment.[2]
- The `phase` function of each cell is adjusted (except when the polarisation of the cell is fixed): the clock of each cell is incremented by $a \cdot n$ clock phases. Here, a is the function parameter that represents the first input $\rightarrow$ last output clock phase difference of the base design.
- If a cell has a number set, this number is incremented by $b \cdot n$. Here, b represents the amount of different numbers of the base design; a base design with A_0 and A_1 is therefore stacked with $b = 2$ to produce A_2 and and A_3.

Additionally, function `stackNTimes` was formed. It takes the amount of times the cell environment should be stacked as input for a more user-friendly experience:

```
stackNTimes :: Int -> (Int,Int) -> [Cell] -> [Cell]
stackNTimes 0 _ = id
stackNTimes n t = stackDesign ( 2 ^ ( n - 1 ) ) t
                . stackNTimes ( n - 1 ) t
```

[2] This assumes the cell environment starts at $z = 0$ and only goes upwards. Compatibility with cell environments with lower z coordinates should be straightforward to implement, though this has not been done as of yet.

4.2.4 Results: an addition calculator

To test the 2^n-bit RCA designs and the stacking function, function `calculate` was designed that asks the user for an expression in the form of $a + b$, and computes the result by constructing the minimal 2^n-bit RCA needed to compute the answer and simulating with the corresponding input to obtain the result. The 2^n-bit RCA is formed by calling `stackNTimes` n `(2,2) ce`. Here, `ce` is the base RCA design version 1 for which a and b are both 2.

Addition with 16-bit numbers produced results in reasonable time when simulated with the max iterations set to 30: around one to two minutes depending on the hardware. The RCA cell environment that computes such sums consists of 65 layers and 481 cells. When computing larger sums, the simulation produces correct results (tested with up to 64-bit numbers) though the simulation time increases exponentially as the size of the cell environment increases drastically.

4.3 Bitwise AND / OR

Bitwise logical operations are an unmissable component of an ALU, used, for instance, to test conditions (x86's `test` instruction). Typically, a bitwise logical operations module would consist of the following bitwise logical operations: AND, OR, XOR and one's complement. We present a simplistic, stackable 1-bit base design for 2^n-bit bitwise AND / OR operations.

The first time a 3MAJ-gate was discussed in this thesis, the method to create AND- and OR-gates from a 3MAJ-gate was also shown (Figure 2.8). The only difference between the two is the polarisation of the fixed cell – this applies to the 1-bit base design of the bitwise AND / OR as well. The stacked design simply becomes a chain of non-coplanar AND- or OR-gates; Figure 4.6 on the next page displays the (once-stacked) bitwise AND.

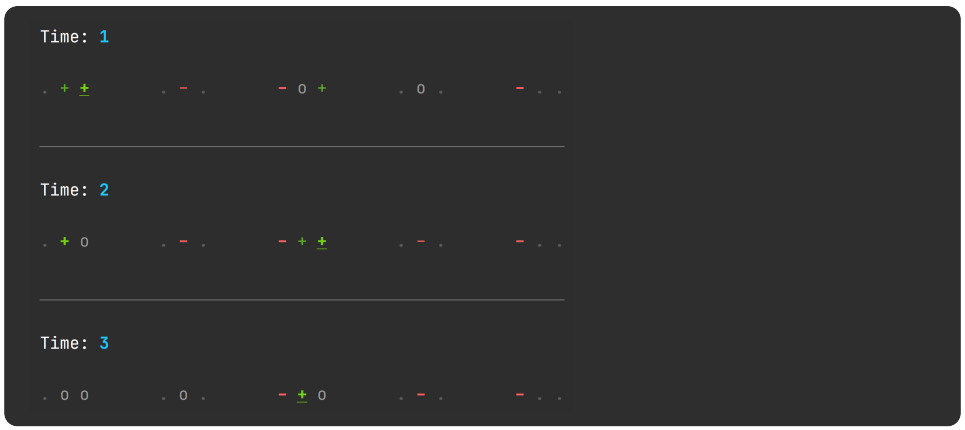


Figure 4.5: Visual simulation output of the bitwise AND, showing $1 \wedge 1$.

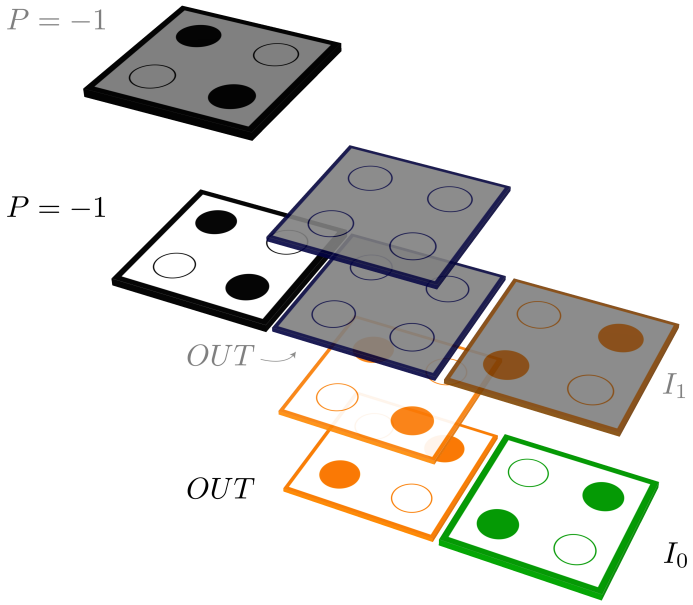


Figure 4.6: A 3D representation of the proposed stackable AND-gate.
The base design consists of an input cell (I_0), an output cell OUT and a regular cell above it, and a cell with a fixed polarisation. If this fixed cell is set to be polarised to $P = -1$ (as shown), the gate computes the bitwise AND of its inputs. If set to $P = +1$, the gate behaves according to a bitwise OR.
The greyed out cells represent the new stack, this introduces a new input cell such that the once-stacked design outputs $I_0 \wedge I_1$. The output cell, which has the `propagate` flag set to true, is moved up by two layers, with the old output cell converting to a regular cell.
Input cell I_0 is polarised to $P = +1$ in the state shown; this polarises the output cell to $P = +1$. Stacked once with input cell I_1 polarised to $P = +1$ as shown, the new output cell is again polarised to $P = +1$. This is shown in Figure 4.5; this figure displays the state at time $T = 1$.

In an n-bit bitwise AND, the output signal is continuously propagated upwards through the individual AND-gates and is ultimately output as $O_{n-1} \wedge I_{n-1}$ at the nth AND-gate, where O_n is the value of the output signal after the nth AND-gate.[3] The exact same logic applies to the n-bit bitwise OR, which outputs $O_{n-1} \vee I_{n-1}$.

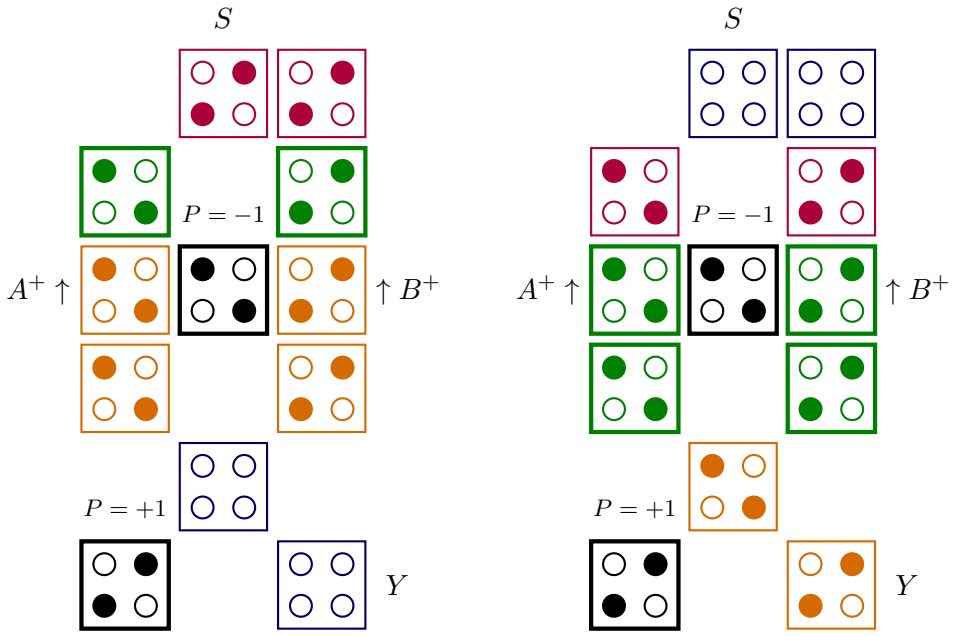
[3]There are methods to compute the bitwise AND / OR in $\log(n)$ steps, hence this linear method of scaling does not produce an optimally efficient gate. This gate is shown as a different example of a linearly stackable design. Given the right scaling algorithm, the design – be it with minor tweaks – can be scaled arboreally to produce a more efficient output. This can be derived from the fact that each two branches, B_1 and B_2, can be merged to one branch, R, with the same operation: $R = B_1 \wedge B$, for example. Achieving a general arboreal scaling solution that can be applied to, for instance, both this gate and the one discussed in the next section is a future goal. More on this in Section 4.4.2.

4.4 Multiplexer

The multiplexer is an important gate that takes 2^n DATA inputs and n SELECT inputs with $n > 0$ and outputs the DATA input corresponding to the given SELECT inputs. An example of its use is in the bitwise logic module, which takes a certain amount of input bits along with an opcode. This opcode acts as the SELECT inputs of the multiplexer. It selects which of the DATA inputs – the outputs of the different logical operations – should be output. Within the field of the nanocomputing paradigm, QCA multiplexers are a relatively widely researched topic. [9, 31, 24, 20, 6, 11]

4.4.1 The 2^n-bit multiplexer

The functioning of a two-input multiplexer (2MUX) with DATA inputs A and B, SELECT input S and output Y can be mathematically defined as the following: $Y = A\overline{S} + BS$. This is implemented in the design seen in Figure 4.7. It has been inspired by [24] and by a presentation of the same author on YouTube.



(a) A 2MUX at time T.

(b) The same 2MUX at time $T + 1$.

Figure 4.7: A 2MUX at two consecutive time steps.
The two DATA inputs A and B are positioned on a layer lower than the one shown. Their positively polarised signals are conducted upwards to the cells next to the negative fixed cell. This forms two AND-gates: $A\overline{S}$, BS. The shape on the bottom that resembles the 5-facet of a die functions as a 3MAJ-gate, or rather an OR-gate because of the fixed positive cell.

This design is not linearly stackable: a 4MUX has double the amount of DATA and SELECT inputs, but an 8MUX, compared to a 4MUX, has double the DATA inputs and only one extra SELECT input. This relation continues for increasing multiplexer sizes: each time the amount DATA inputs is doubled, the amount SELECT inputs is incremented by one. The design instead scales similarly to a perfect binary tree: adding a layer to the tree doubles the amount of leaves and increases the height by one.

This relation was inspired by a figure from [20], where 2MUX blocks are connected to each other as nodes in a binary tree shape; each layer of the tree has a distinct SELECT input that is supplied to all multiplexers on that layer. Each 2MUX creates one DATA output from two DATA inputs, effectively defining the property of a reversed (leafs → root) binary tree.

Figure 4.8 below shows the main layer of the proposed base design of the 2^n-bit multiplexer. Essentially we see two 2MUX designs similar to the one shown in Figure 4.7 merged together to form a 4 input - 2 output multiplexer (we write $[4 \to 2]$MUX from now on). The outputs, Y_0 and Y_1, are selected by the second SELECT input (S_1) and are propagated upwards as inputs of the 2MUX from Figure 4.7. Output of this 2MUX is selected by the first SELECT input (S_0); this is the output of the 4MUX as a whole.

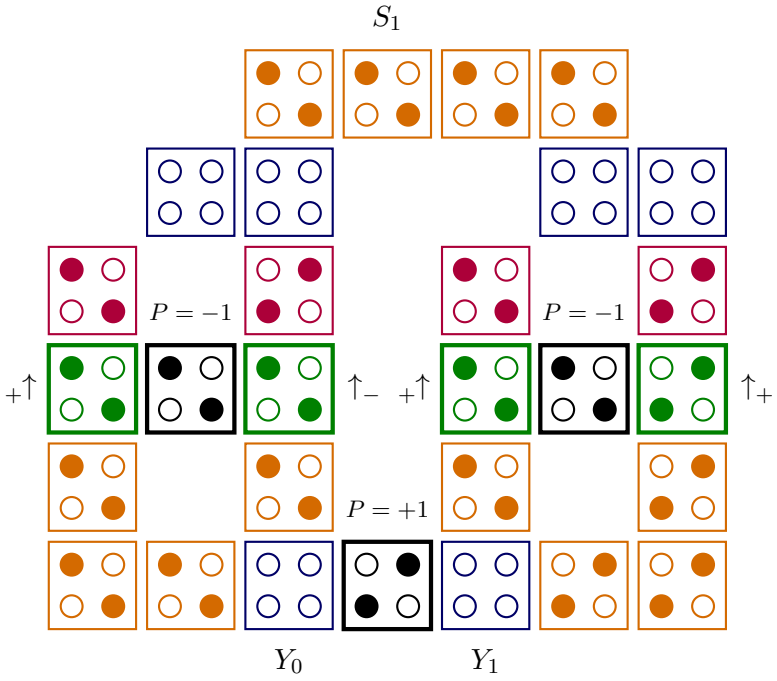


Figure 4.8: The proposed $[4 \to 2]$MUX. The outputs Y_0 and Y_1 are given to the 2MUX from Figure 4.7 as DATA inputs to complete the 4MUX design. The four DATA inputs, similarly to the previous design, are propagated up to the cells next to the fixed negative cells. Their polarisations are shown.

It should be noted that the different clock phases have been generously distributed in the designs shown in the last two figures. This is not optimal when designing a minimum delay QCA, though this design was formed with the intent of creating a stackable base design. Because of this, the stability of the design – which severely decreases with multiple iterations of the stacking algorithm as a result of more unwanted inter-cellular interactions – was favoured over the input-output latency.

In the next section, we present the stacking algorithm that creates a functional 8MUX from the 4MUX base design after one application, a 16MUX after another application and so on. This algorithm was experimentally used to produce cell environments of multiplexers up to 64-bit – rendering these in a console becomes tediously slow as the size of the design increases, though a script was written that converts visual output into an HTML page to aid rendering times (Section 4.4.4). Successful simulations have been done with 32MUX, though this has not been achieved with the 64MUX. The limitations concerning this will be discussed in Section 4.4.3.

4.4.2 The binary tree stacking algorithm

Much of this stacking algorithm shares properties with a binary tree. Our base design, the 4MUX discussed in the previous paragraph, can be compared to a perfect binary tree with three nodes and four leaves. The leaves represent the inputs; they are processed by the $[4 \to 2]$MUX layer shown in Figure 4.8. The two nodes at depth 1 of the binary tree represent this multiplexer. The multiplexer is essentially two 2MUXs merged together into one design; they both get SELECT input S_1 – note that the number matches the depth. The 2MUX at the upper layer, shown in Figure 4.7, is the root node at depth 0 and has SELECT input S_0.

The algorithm consists of three steps. First we form the new lowest multiplexing layer, similar to increasing the height of our perfect binary tree by one. Applied to our 4MUX base design, this new layer becomes a $[8 \to 4]$MUX with SELECT input S_2, on top of which the 4MUX is stacked. Secondly, we create a layer consisting of wires connecting the outputs of the new multiplexing layer to the inputs of the previous stack. This step was not required to connect the multiplexers from Figures 4.7 and 4.8, since the outputs of the former fit directly under the inputs of the latter. New layers, however, do not share this property with the layer they should connect to. Lastly, we assign clock phases to the design in a way that makes the design stable without adding too many unnecessary clock phases. Especially this is a difficult task to generalise for each stacking operation, as the aforementioned criteria affect a small design differently than a large one.

We briefly discuss each point in the following paragraphs.[4]

[4]Much detail is left out. The algorithm was implemented in just over 100 dense lines of Haskell code, whereas the core of the simulation engine required around half of that.

Creating the new lowest multiplexing layer

This step is conceptually the simplest of the three. Essentially, we take the lowest multiplexing layer – let this be a $[2a \to a]$MUX – and copy it twice such that the new lowest multiplexing layer consists of two $[2a \to a]$MUXs to form a $[4a \to 2a]$MUX. The original SELECT inputs are converted to regular cells and instead a new row of cells that includes the new SELECT input is added to the top of the layer, connecting the separate multiplexers. This is also applied in the design seen in Figure 4.8, where the row containing S_1 connects to the SELECT input rows of the individual 2MUXs. For the purpose of stability and consistency, each row is offset by one clock phase. Resultantly, the SELECT input row of the new layer becomes the first clock phase each time, offsetting the rest of the layer by one clock phase. Clock offsets for the rest of the design are discussed in the upcoming paragraph on clock phases.

The tricky part of this step is determining the length and starting column of this new row. The function that is ultimately called to stack the multiplexer design is named `stackTreeDesign` and, equivalently to function `stackNTimes`, takes an `Int`-type argument n that is incremented by one with each additional stacking, starting at $n = 1$. This parameter is, amongst other things, used to compute the aforementioned properties of the new row, and in it, the position of the new SELECT input.

The $[4 \to 2]$MUX has a horizontal length of 7. Leaving one cell space in between, two of these multiplexers next to each other would add up to a horizontal length of 15. This sequence continues to 31, 63, et cetera; clearly these are all powers of two minus one. When function `stackTreeDesign` is called for the first time on the 4MUX base design with $n = 1$, it generates the new lowest multiplexing layer with a horizontal length of $2^2 \cdot 2^n - 1 = 15$. The new SELECT input s is positioned in the same column as the spacer between the two $[4 \to 2]$MUXs: the $2^2 \cdot 2^{n-1} = 8$th column. Since x values start at zero, this translates to: $s_x = 4 \cdot 2^{n-1} - 1$.

The new row always has one cell fewer on the left side of the SELECT input than on the right side. It connects to the two rows under it by beginning next to the rightmost cell of the first row and ending next to the leftmost cell of the second row. This leads to somewhat irregular row lengths; hence there is no straightforward expression that tells us the x value of the first cell in the row (r_x^0) for a given n. The first six values of the emergent sequence are as follows: $\left[r_x^0(1), r_x^0(2), \ldots, r_x^0(6)\right] = [5, 10, 21, 42, 85, 170]$. A linear homogeneous recurrence relation that produces the sequence was constructed:

$$\forall n \in \mathbb{N} : n \geq -1 \to r_x^0(n) = \begin{cases} 1, & \text{if } n = -1. \\ 2, & \text{if } n = 0. \\ 2^{n+1} + r_x^0(n-2) & \text{otherwise.} \end{cases} \tag{4.5}$$

The associated solution of this relation was computed, this yielded a small yet notable performance increase.

$$r_x^0(n) = \frac{-3 + (-1)^{n+1} + 2^{n+3} \cdot (3 + (-1)^{2n+1})}{6} \tag{4.6}$$

Since the rightmost cell of the new row is in the same column as the leftmost cell of the row that comes after it with the next stacking operation, we can express the ordered sequence of x values of the cells in the new row as follows:

$$\left[r_x^0(n), r_x^0(n) + 1, \ldots, r_x^0(n+1)\right].$$

Forming the connecting wires

In the previous paragraph, we discussed how a $[4a \rightarrow 2a]MUX$ can be formed from a $[2a \rightarrow a]$MUX. In this paragraph, we tackle the problem of connecting the $2a$ outputs from the former to the $2a$ input from the latter. We describe the algorithm that produces the layer with the connecting wires, and additionally the modifications to the rest of the cell environment that are required to fit this layer in.

The algorithm, simply put, creates Manhattan paths from each output of the $[4a \rightarrow 2a]$MUX to the corresponding input of the $[2a \rightarrow a]$MUX, whilst maintaining a minimum wire spacing of one cell; this is said to be sufficient to avoid cross-wire interference. [17] This is done in such a way that the right half of the wires is a mirror copy of the left half; therefore we only need to actively create the wires for one half.

In order to enable this mirroring property as well as the creation of the Manhattan paths, the cell environment given as input to the stacking algorithm should be centred with respect to the new lowest multiplexing layer. From now on, we refer to the former as the root and to the latter as the new leaf. The root is shifted to the right by 2^{n+1} to perfectly align the centres of the root and the new leaf. By design of the stacking algorithm, each of the multiplexing layers is vertically aligned in such a way that the SELECT input row is on the very top of the design.[5]

The paths are created starting with the left outermost path; this one goes up from the leftmost output of the new leaf until it reaches the height of the inputs of the root.[6] Any cells that should be put horizontally to connect this part of the path to the leftmost input of the root are added to the path: this completes the first path.

The other paths follow a slightly different logic, although the function that creates paths is the same for each path. A path goes up from the new

[5] For clarification: in this context, vertically directional words like up/down, high/low and top/bottom refer to the two-dimensional representation of a layer.

[6] The output cells locations of the new leaf are obtained by filtering the cell environment for cells with the `propagate` flag set to `True`. This flag is strictly set for the output cells of the lowest multiplexing layer.

leaf output until there is exactly one vertical cell space between the end of the path and the first path. The path then goes to the right until it reaches the same column as the root input that the path should get to; it goes up to that input to complete the path. If a new leaf output is on or directly adjacent to an already existing path, the path starts at a lower point, exactly one cell space removed from the lowest existing path in the column. This makes the path misalign with the corresponding new leaf output; this is fixed with the method that is discussed next. The left half of the wires is now in its elemental form.

The y value of the lowest path directly determines the new y value of the new leaf outputs. All other paths are extended such that they now begin at this y value. The new leaf is also modified such that each of the output cells is moved to this new y value and a connecting wire from the old position straight down to the new position is made. These extensions can be seen in Figure 4.9 on the next page.

Two other figures are shown on this page; they all show a part of the cell environment generated by two applications of the tree stacking algorithm on the 4MUX base design. The figures are shown at different times in such a way that the output signals of the [16 $\rightarrow$ 8]MUX in Figure 4.9 can be followed to the [8 $\rightarrow$ 4]MUX in Figure 4.11. The signals take multiple clock phases to pass through the layer with the connecting wires, shown in Figure 4.10. The consideration for this is expounded in the next paragraph.

Setting the clock phases

We previously discussed that all cells that are not in the new leaf should be offset by one clock phase by the addition of the new SELECT input row. Furthermore, cells in the root should get an additional phase offset in accordance with the clock delay of the wire layer. A number of rules that produce this delay were defined in order to yield consistent signal transmission through the wires for multiplexers up to 32-bit, whilst keeping delay relatively minimal. Issues regarding the stability of higher bit multiplexers are discussed in Section 4.4.3.

The programmatical setting of clock phases for a wire layer *of any size* is a difficult task when considering both the stability of the wires – the prevention of crosstalk between wires – and the total clock delay – the minimisation of the clock phases required to consistently transduce signals through the wires. Parallel wires spaced one cell space apart are generally stable, though wire corners turned out to be prone to inconsistent signal transduction and thus require additional clock phases.[7]

[7] It is possible that this phenomenon occurs by cause of QCA-STACK's method of simulation, and especially the cell evaluation order. Due to time restrictions, a comparative analysis with simulation output of QCADesigner was not done, though this would confirm the validity of the concern. More on this in Section 4.4.3.

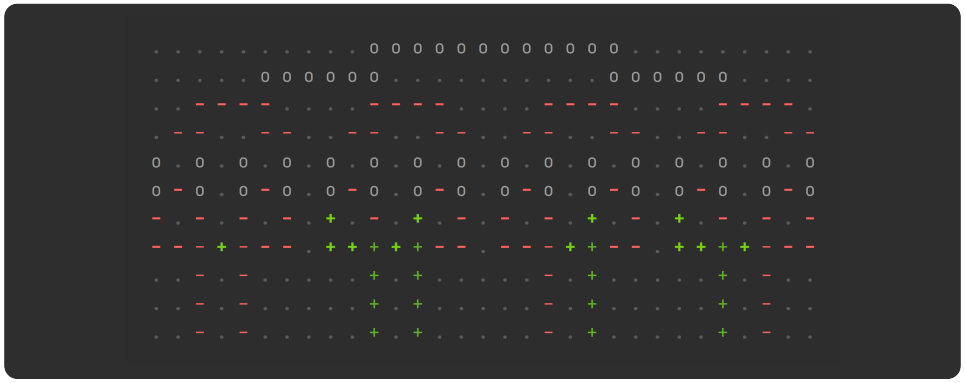


Figure 4.9: A $[16 \to 8]$MUX, created with the tree stacking algorithm.

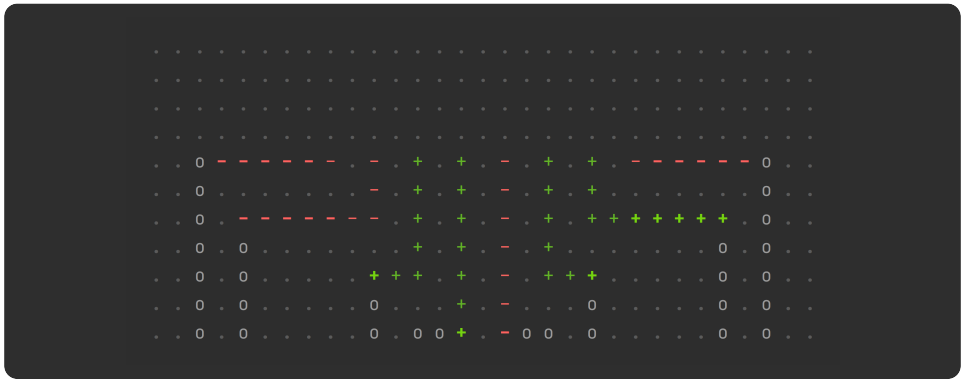


Figure 4.10: The layer that connects the $[16 \to 8]$MUX to the $[8 \to 4]$MUX.

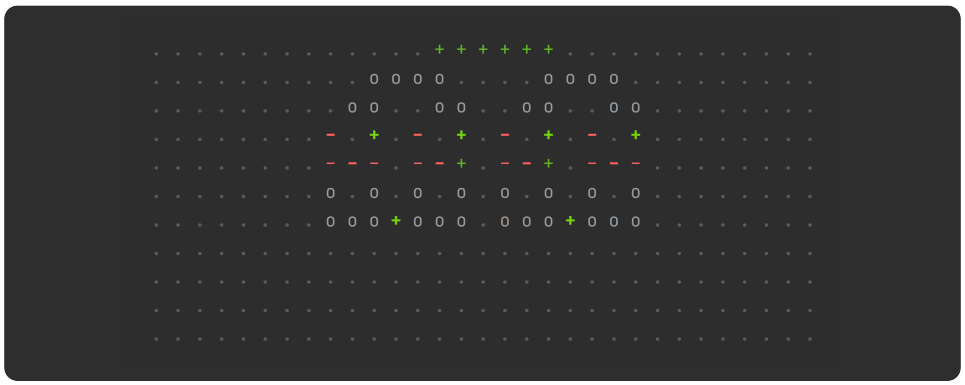


Figure 4.11: A $[8 \to 4]$MUX. Here, it passes the rightmost output of each consecutive output pair of the $[16 \to 8]$MUX on to the $[4 \to 2]$MUX.

When the multiplexer is scaled up, each wire receives more external influence; this complicates the process of defining a layer of wires that functions consistently with a minimum amount of clock delay. Again, the function is applied to wire layers of any size, hence, for instance, a rule that fixes a stability issue for the 32MUX could introduce unnecessary delay in the 16MUX.

The clocks for each cell in the wires are set by sequentially traversing the wires cell by cell, starting from the bottom each time. If a certain condition concerning a relative position of a nearby cells in a different wire is met, the clock is increased for the rest of the wire. When all wires have been fully traversed, the clocks are synced up such that each wire has the same delay. This introduces a bottleneck factor: the wire with the highest clock delay decides the delay for the other wires.

For debugging purposes of the wire layers, the function `showCEClocks` was designed. It makes a call to the function that is called at every time step when simulating with function `runSimVisual`, but gives it a flag that makes the function show just the clock numbers of cells in the given cell environment. Cells that are in the Switch phase at time $T = 0$ are displayed with '0', cells in the Switch phase at time $T = 1$ with '1', et cetera. Output of this function is seen in Figure 4.12 below.

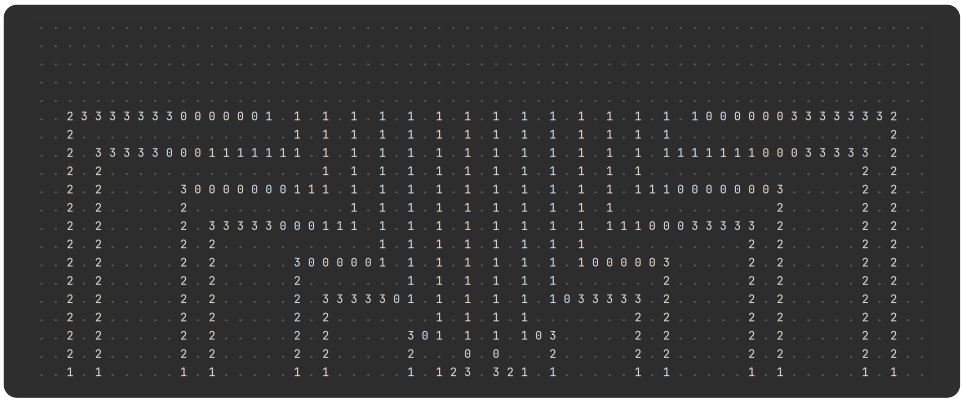


Figure 4.12: Output of function `showCEClocks`, showing the wire layer that connects the outputs of the $[32 \rightarrow 16]$MUX to the inputs of the $[16 \rightarrow 8]$MUX of the triply stacked 4MUX base design.

A general tree stacking algorithm?

The algorithm that is discussed in this section was specifically designed to be applied on the proposed 4MUX base design, however no significant constants were used.[8] Other functions sharing the arboreal scaling property

[8]For example: the algorithm sets the `label` field to `"SEL"` for the new SELECT input.

of the multiplexer exist: the more efficient approach to scaling the bitwise AND / OR, discussed in the footnote of Section 4.3, is one such example. We speculate that the proposed algorithm could scale any design that scales as a binary tree, given that the algorithm is parametrised with scaling properties that are specific to the design. Examples of this for our multiplexer are the functions s_x and r_x^0 and the horizontal root offset function that is used to centre the root with respect to the new leaf. Transforming the algorithm into a general N-tree stacking algorithm might not be a far reach either.

4.4.3 Limitations and future work

In earlier paragraphs, it was mentioned that the stability of the stacked multiplexer design decreases from sizes 64-bit and on. With every scale-up, each part of each wire becomes longer and the number of wires are increased. This leads to a less stable design; wires have more space to lose the integrity of the signal they are carrying.

There is a simple solution to this: more clock phases. This approach was applied to get the 32MUX stable, though the effort was not continued for the 64MUX. The additional clock-setting rules that would be needed for this would most likely impair the less-stacked cell environments resulting from the `stackTreeDesign` function in terms of clock delay. Alternatively, the clocks in each wire could be incremented after every n cells. This is guaranteed to work for $n = 1$ although this results in a massive clock delay. Setting $n = 5$ could produce stable designs for higher-bit multiplexer with reasonable clock delay, though this was not verified.

Optimising the clock setting algorithm is an interesting topic for future research. It would contribute to a resilient tree scaling algorithm, to be used to form a 64MUX, 128MUX or even beyond. After searching for publications that present a 16-bit multiplexer or higher, it was concluded that the effort to create a design of this scale has not been made, or at least no such effort was published. However, certain publications do propose a scheme to create a multiplexer of this order, or present a lower-bit multiplexer and advertise it to be used as a building block for higher-bit one. [20, 9]

As discussed in footnote 7 of this chapter, QCA-STACK's method of simulation could be to blame concerning the wire instability at higher-bit multiplexers. A script that converts a QCA-STACK cell environment to a QCADesigner input file can be made to test whether this continuous simulation engine produces different results. If QCADesigner shows that wires that are only stable in QCA-STACK after adding a certain amount of clock phases are in fact stable with fewer clock phases, this would lead to two subjects for future work. Firstly, the clock setting algorithm could be made more conservative with adding clock phases, and secondly, our discrete simulation engine would need attention in order to be consistent with QCADesigner's output; it most likely requires a different evaluation order algorithm.

Lastly, a note must be made on the QCA-STACK's input system. With the large generated multiplexers, the delays between some of the inputs – specifically the SELECT input S_0 compared to the DATA inputs in a 16- or higher-bit multiplexer – become significant, preventing the input system to supply the user-given inputs to the QCA in the expected logical manner. Possibly, the input system can be extended to process extra user input, specifying certain clock delays. Alternatively, an algorithm could be made that traverses a QCA to algorithmically determine the clock delay between two cells, which could be used to determine the required clock offset of certain inputs.

4.4.4 Visual output to HTML

As cell environments are scaled up to the order of the algorithmically generated 32MUX or 64MUX – respectively consisting of 1359 cells and 19 layers, and 4503 cells and 23 layers – viewing visual simulation output in a console becomes infeasible. Even faster systems are unable to render such cell environments in an instant; add this to the fact that the procedurally generated multiplexers of this specific size respectively require around 30 and around 50 clock phases to produce a result, and it becomes clear that an alternative method of displaying simulation output must be sought after for the visual assessment of the intended functionality of the QCA. Most CLIs also limit the number of consecutive lines that can be viewed, preventing the user to see the entire simulation output for larger queries.

To improve the user experience in this regard, a method was devised that enables the user to view visual QCA-STACK output in a browser as a HTML page. The simulation running function `runSimNPhasesWrite` is parametrised with a T_{max} – described in Section 3.3.1 – and writes the visual simulation output (including ANSI escape sequences) to an output file. This file is converted to an HTML file with the use of a Python script that was created for this purpose, which depends on the library `ansi2html`.[9]

Both the intended layout and the intended colours are retained in the HTML output. Browser viewing offers much faster rendering times and has no limits regarding the size of the page. The script along with exemplary multiplexer simulation outputs as HTML pages are found on the QCA-STACK GitHub page.[10]

This method of viewing simulation output improves over CLI output in a number of ways, although the rendering times are still not ideal for large designs. For larger scale use, a method that enables hardware accelerated viewing should be created in order to achieve optimal rendering times.

[9] https://github.com/pycontribs/ansi2html
[10] https://github.com/wlambooy/QCA-STACK

Chapter 5

Related Work

In 1993, Lent et al. published an article that became known as the conception of the QCA paradigm. [15] In this article, they present the fundamental physics of the system as well as small AND- and OR-gates and methods for QCA memory storage. The system of clocking QCAs is later worked out in 2001 by Hennessy and Lent. [10] Tóth later writes a dissertation in which the physics and the clocking system are extensively described. [27] This thesis is used as a foundational reference work for the simulation engines of QCADesigner, presented in a paper in 2005. [30] QCADesigner has been used in numerous publications to verify the functioning of designs presented, including each reference in this thesis that presents a non-schematical design.

It is particularly interesting to compare this work with publications presenting RCAs or multiplexers. The RCA design we present can be used to generate RCAs exceeding 512 bits with ease, though successful simulations have only been achieved with an up to 64-bit RCA due to limitations of the hardware used for the simulation – the amount of memory that is needed scales exponentially in the current state of QCA-STACK. Conversely, other researchers were able to simulate a 128-bit RCA. [4] If we compare the cell count of this design with our procedurally generated design, however, we notice a significant difference. The 128-bit RCA design consists of 32256 cells, while our generated design consists of just of a little more than a tenth of that: 3841 cells. A generated 1024-bit RCA consists of 30721 cells, fewer still than the existing 128-bit RCA.

As stated in Section 4.4.3, no publication was found that presents a 16:1 multiplexer. We can compare the cell count of an existing 8:1 multiplexer to our procedurally obtained version, however. The early generation of such multiplexers consist of around 500-700 cells, though newer designs have been presented that are able to achieve the same with just over 300 cells. [22, 14] After searching for the lowest cell count of an existing multiplexer of this capacity, it was concluded that the lowest cell count is 260 cells, presented in [9]. In comparison, our 8:1 multiplexer consists of 165 cells.

Chapter 6

Conclusions

In the previous chapters, we have seen how the QCA system was reduced to its functional core to form the QCA-STACK simulation engine, and how it was extended with stacking algorithms that were able to generate extensible 2^n-bit ALU components. Now we discuss how this concept can be used to ultimately form a 2^n-bit nano-scale processing cuboid.

Consider the design of an extensible 2^n-bit ALU. This ALU would consist of a 2^n-bit RCA, a 2^n-bit bitwise logical operations module, a 2^n-bit bit-shift, et cetera. All of these designs would be stackable and each design has a specific function or function call that transforms it into a 2^{n+1} design. The ALU connects the designs in such a way to create a functional 2^n-bit unit. Now extending this unit into a 2^{n+1}-bit one simply requires each individual component to be stacked using the corresponding stacking operation. We have seen that linearly stackable components require no extra space on the plane when stacked, though arboreally stackable components do. Undoubtedly, an algorithm can solve this spacing issue for us. Additionally, an algorithm should be designed that connects the components and synchronises the clock phases to produce a 2^{n+1}-bit ALU.

In theory, the same logic could be applied to other required parts of a processor, forming a processor consisting of densely packed components that would look similar to skyscrapers in Downtown New York. Hence, by the three-dimensional nature of the structure, a scalable 2^n-bit nano-scale processing cuboid emerges.

6.1 Future works

Realising the proposed concept requires certain algorithms whose processes have not been considered throughout this thesis, leaving much work to be done. Additionally, the correctness of the discrete simulation engine should be assessed and possibly improved, as discussed in Section 3.1.3. Also the tree stacking function requires attention; this was discussed in Section 4.4.3.

Bibliography

[1] Adiabatic theorem. https://en.wikipedia.org/wiki/Adiabatic_theorem. Last accessed on 20 July 2021.

[2] Moore's law is dead, says gordon moore. https://web.archive.org/web/20190723141141/https://www.techworld.com/news/tech-innovation/moores-law-is-dead-says-gordon-moore-3576581/. Last accessed on 19 July 2021.

[3] Firdous Ahmad, Gm Bhat, Hossein Khademolhosseini, Saeid Azimi, Shaahin Angizi, and Keivan Navi. Towards single layer quantum-dot cellular automata adders based on explicit interaction of cells. *Journal of Computational Science*, 16:8–15, 02 2016.

[4] N. Archana, M. Chandana, M. Lakshmi, M. Nandini, and K. Praveena. A novel 128 bit adder using qca. *International Journal of Engineering Sciences & Research Technology*, pages 407–4012, 06 2014.

[5] Milad Bagherian Khosroshahy, Mohammad Moaiyeri, Keivan Navi, and Nader Bagherzadeh. An energy and cost efficient majority-based ram cell in quantum-dot cellular automata. *Results in Physics*, 7, 09 2017.

[6] Amir Mokhtar Chabi, Samira Sayedsalehi, Shaahin Angizi, and Keivan Navi. Efficient qca exclusive-or and multiplexer circuits based on a nanoelectronic-compatible designing approach. *International Scholarly Research Notices*, 2014:463967, Oct 2014.

[7] Mojtaba Divshali, Abdalhossein Rezai, and Asghar Karimi. Towards multilayer qca siso shift register based on efficient d-ff circuits. *International Journal of Theoretical Physics*, 57, 11 2018.

[8] Davide Giri, Marco Vacca, Giovanni Causapruno, Maurizio Zamboni, and Mariagrazia Graziano. Modeling, design and analysis of magnetoelastic nml circuits. *IEEE Transactions on Nanotechnology*, 15:1–1, 11 2016.

[9] Sara Hashemi, Mostafa Rahimi Azghadi, and Ali Zakerolhosseini. A novel qca multiplexer design. pages 692 – 696, 09 2008.

[10] K. Hennessy and C. Lent. Clocking of molecular quantum-dot cellular automata. *Journal of Vacuum Science & Technology B*, 19:1752–1755, 2001.

[11] Jun-Cheol Jeon. Designing nanotechnology qca–multiplexer using majority function-based nand for quantum computing. *The Journal of Supercomputing*, 77, 02 2021.

[12] Upal Joy, Shourov Chakraborty, Sumaiya Tasnim, Mohammad Hossain, Abdul Siddique, and Mehedi Hasan. Design of an area efficient quantum dot cellular automata based full adder cell having low latency. 01 2021.

[13] Angshuman Khan and Rajeev Arya. Optimal demultiplexer unit design and energy estimation using quantum dot cellular automata. *The Journal of Supercomputing*, 77, 02 2021.

[14] M. Kianpour and R. Sabbaghi-Nadooshan. Optimized design of multiplexor by quantum-dot cellularautomata. *International Journal of Nanoscience and Nanotechnology*, 9(1):15–24, 2013.

[15] C S Lent, P D Tougaw, W Porod, and G H Bernstein. Quantum cellular automata. *Nanotechnology*, 4(1):49–57, jan 1993.

[16] Jeyalakshmi Maharaj and Santhi Muthurathinam. Effective rca design using quantum dot cellular automata. *Microprocessors and Microsystems*, 73:102964, 2020.

[17] Chiradeep Mukherjee, Saradindu Panda, Asish Kumar Mukhopadhyay, and Bansibadan Maji. Introducing Galois field polynomial addition in quantum-dot cellular automata. *Applied Nanoscience*, 9(8):2127–2146, November 2019.

[18] Keivan Navi, Razieh Farazkish, Samira Sayedsalehi, and Mostafa Rahimi Azghadi. A new quantum-dot cellular automata full-adder. *Microelectronics Journal*, 41(12):820–826, 2010.

[19] Samuel Sze Hang Ng, Jacob Retallick, Hsi Nien Chiu, Robert Lupoiu, Lucian Livadaru, Taleana Huff, Mohammad Rashidi, Wyatt Vine, Thomas Dienel, Robert A. Wolkow, and Konrad Walus. Siqad: A design and simulation tool for atomic silicon quantum dot circuits. *IEEE Transactions on Nanotechnology*, 19:137–146, 2020.

[20] Veerendra Nune and Addanki Purna Ramesh. Novel design of multiplexer and demultiplexer using reversible logic gates. *International Journal of Engineering and Technology(UAE)*, 7:80–87, 08 2018.

[21] Azzurra Pulimeno, Mariagrazia Graziano, Alessandro Sanginario, Cauda Valentina, Danilo Demarchi, and Gianluca Piccinini. Bis-ferrocene molecular qca wire: Ab initio simulations of fabrication driven fault tolerance. *IEEE TRANSACTIONS ON NANOTECHNOLOGY*, 12:498–507, 07 2013.

[22] Hamid Rashidi, Abdalhossein Rezai, and Sheema Soltany. High-performance multiplexer architecture for quantum-dot cellular automata. *Journal of Computational Electronics*, 15, 09 2016.

[23] Hamid Roshany and Abdalhossein Rezai. Novel efficient circuit design for multilayer qca rca. *International Journal of Theoretical Physics*, 58, 06 2019.

[24] Soudip Sinha Roy. A novel optimized multiplexer design in quantum-dot cellular automata. *International journal for research in applied science and engineering technology*, 5, 2017.

[25] Amanpreet Sandhu and Sheifali Gupta. Performance evaluation of an efficient five input majority gate design in qca nanotechnology. *International Journal of Reconfigurable and Embedded Systems (IJRES)*, 8:194, 11 2019.

[26] F. Sill, Philipp Niemann, R. Wille, and R. Drechsler. Near zero-energy computation using quantum-dot cellular automata. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16:1 – 16, 2020.

[27] Géza Tóth. *Correlation and coherence in quantum-dot cellular automata*. PhD thesis, University of Notre Dame, 2000.

[28] Marcel Walter, Robert Wille, Daniel Große, Frank Sill Torres, and Rolf Drechsler. An exact method for design exploration of quantum-dot cellular automata. pages 503–508, 03 2018.

[29] Marcel Walter, Robert Wille, Frank Sill Torres, Daniel Große, and Rolf Drechsler. Scalable design for field-coupled nanocomputing circuits. 01 2019.

[30] K. Walus, T.J. Dysart, G.A. Jullien, and R.A. Budiman. Qcadesigner: a rapid design and simulation tool for quantum-dot cellular automata. *IEEE Transactions on Nanotechnology*, 3(1):26–31, 2004.

[31] Li Xingjun, Shao Zhiwei, Cheng Hongping, and Mohammad Reza Jamali Haghighi. A new design of qca-based nanoscale multiplexer and its usage in communications. *International Journal of Communication Systems*, 33(4):e4254, 2020. e4254 IJCS-19-0799.R1.

Appendix A

Input Files

A.1 RCA: Version 2

```
= = = = = = = =
        c
        0
      0 1 y
a 0 1 1 y 1 s
      0 1
        0
        b
= = = = = = = =
      0 0
```

```
0
0
```

```
      0 0
= = = = = = = =
      0
      0
      0
      1
0 1 1 1 1 1 1
      1
      0
= = = = = = = =
```

```
                1
                1
```

```
= = = = = = = =
```

```
        S
        2
      x x
A 1 2 2 2 1 1
      1 2 1
        1
        B
= = = = = = = =
```

```
1             1
1             1
```

```
        1 1
= = = = = = = =
          C
          2
          2
          2
1 1 1 1 2 1 1
          1
          1
= = = = = = = =
        2 2
```

```
= = = = = = = =
```

```
$

a:
- input
- label = "A"
- clock = 0
- number = 0

b:
- input
- label = "B"
- clock = 0
- number = 0

A:
- input
- label = "A"
- clock = 1
- number = 1

B:
- input
- label = "B"
- clock = 1
- number = 1
```

```

c:
- input
- label = "Cin"
- clock = 0
- propagate

C:
- output
- label = "Cout"
- clock = 2
- number = 1
- propagate

s:
- output
- label = "SUM"
- clock = 1
- number = 0

S:
- output
- label = "SUM"
- clock = 2
- number = 1

y:
- clock = 1
- offset = (0,-0.5,0)

x:
- clock = 2
- offset = (0.5,0,0)
```

A.2 AND / OR

```
= = =
  o i
= = =
  1
= = =
-
= = =

$

i:
- input
- label = "I"
- clock = 0
- number = 0

o:
- output
- label = "AND"
- clock = 1
- propagate
```

```
= = =
  o i
= = =
  1
= = =
+
= = =

$

i:
- input
- label = "I"
- clock = 0
- number = 0

o:
- output
- label = "OR"
- clock = 1
- propagate
```

A.3 Multiplexer

```
= = = = = = =

a   b   c   d

= = = = = = =

2   2   2   2

= = = = = = =
    0 S 0 0
  1 1     1 1
2   2   2   2
3 - 3   3 - 3
4   4   4   4
4 4 p + p 4 4
= = = = = = =
```

```
    5   5
= = = = = = =
    1   1
    5   5
    5   5
= = = = = = =
    1   1
      s 0
    1   1
    2 - 2
    2   2
      3
    +   o
= = = = = = =
```

```
$

a:
- input
- label = "I"
- clock = 2
- number = 0

b:
- input
- label = "I"
- clock = 2
- number = 1

c:
- input
- label = "I"
- clock = 2
- number = 2

d:
- input
- label = "I"
- clock = 2
- number = 3
```

```

s:
- input
- label = "SEL"
- clock = 0
- number = 0

S:
- input
- label = "SEL"
- clock = 0
- number = 1

o:
- output
- clock = 3
- label = "OUT"

p:
- clock = 5
- propagate
```