Memory Architecture for Quantom-dot Cellular Automata

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Computer Science and Engineering

by

Sarah Elizabeth Frost, B.A.

Dr. Peter Kogge, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
March 2005

Memory Architecture for Quantom-dot Cellular Automata

Abstract

by

Sarah Elizabeth Frost

Quantum-dot Cellular Automata (QCA) is a novel nanotechnology with great potential for very dense memory and low power logic. This work presents the H-memory architecture, a memory architecture that exploits the characteristics of QCA and results in order of magnitude density gains over end of the roadmap SRAM and DRAM. Two enhancements to the basic architecture are also presented, including a complete merging of processor and memory. Finally, a novel clocking wire layout is presented and its effect on architecture discussed.

This thesis is dedicated to my parents, Thomas and Linda. Thank you for teaching me by word and example to pursue personal excellence and for your encouragement, advice, and understanding. Thanks for enjoying my adventures with me.

CONTENTS

FIGURES

ACKNOWLEDGMENTS

I would like to especially thank my advisor, Dr. Peter Kogge, for guiding me onto a path of valuable and fun research, for providing incredible opportunities to learn and grow as a person and a researcher, and for allowing me to pursue a few detours that were more valuable and horizon broadening than I can ever say.

I would also like to thank my committee, Dr. Eugene Henry and Dr. Sharon Hu for their valuable contribution to this endeavor.

Many thanks also go to Dr. Craig Lent, Mike Niemier, and Arun Rodrigues for valuable discussions about the nature of QCA and for turning the task of designing with QCA from looking like a snarling, angry bear to a good natured puppy. A particular thanks to Dr. Megan Frost for explaining to this non-chemist the chemistry and principles of how the molecular world works. Thank you to Megan, Mike, and Arun for showing by example what grad school and research are all about.

A very special thanks to Richard Murphy for being a valuable sounding board, giving invaluable advice and encouragement, and keeping me as sane as possible during this pursuit.

Finally, I would like to thank the Semiconductor Research Corporation (SRC) and the Clare Boothe Luce Foundation for supporting this work.

CHAPTER 1

Introduction

1.1 The Problem

The goal of computer designers and manufacturers is to produce smaller, faster computers. In 1965, Gordon Moore described the success of the industry in this matter noting that between 1959 and 1965, the number of components on a die grew exponentially [37]. This trend has continued with the number of transistors on a die nearly doubling every 18-24 months. This success has been achieved primarily by shrinking the size of the transistor, aided by the increasing size of the die. For instance, Intel's 4004 released in 1971 was made of 2300, 10 micron transistors on a 12 mm^2 die [57]. In contrast, today's chips contain tens or hundreds of millions of transistors near 0.07 microns on dies on the order of several hundred square millimeters.

However, the current strategy of shrinking the transistors and maintaining the same design paradigm will soon be insufficient to meet physical, economic, and architectural barriers. The smallest transistors in production today operate despite quantum effects. In the near future, the operation of transistors will be dominated by the quantum world. The current device, the CMOS transistor, will need to be replaced by one that embraces these quantum effects and takes advantage of the physics that governs at the nano-scale. Fabrication costs, short lifetime of chip generations, rising capital costs, and demand for computing power from consumers all create economic challenges for the semiconductor industry [55]. Finally, as the gap between processor and memory speeds continues to grow, the von Neumann bottleneck will create a greater and greater architectural barrier to continued performance increases.

In addition to these general comments, memory architecture in particular needs to be reexamined for two reasons. First, the gap between memory access times and processor cycle times is large and growing faster (figure 1.1). Most recent architectural advancements in processor architecture are aimed at masking the memory access latency. As the relative divide between memory access time and processor cycle time grows, it will become more and more difficult for processor architectures to mask this latency. Rather than attempting to mask this latency, it is time to address memory architecture to reduce the latency itself.

Second, the transistor paradigm will have particular trouble in memory at the nanoscale. As transistors are scaled down in size, the leakage current increases. This is a particular problem for memory cells. Leakage current translates to heat generation which limits the density of storage. If memory cells are too close together, the heat generated could destabilize the cells (e.g. flip a stored zero to a one). This constraint will negatively impact the memory density of CMOS memory at the nanoscale.

Finally, the array architecture does not translate well to the nanoscale due to an increase in both transient and permanent errors. Transient faults will be more common because the energies at which bits are stored will be lower than current memories and will therefore be more susceptible to fluctuations due to doping problems both at fabrication time and during use caused by electromigration of atoms during memory operation. Bits will also be more susceptible to high energy particles which cause energy spikes that could flip stored bits. Permanent errors will also be more common because of the difficulty of fabrication at the nanoscale. While massive redundancy could address the permanent errors as the extra rows in today's array memories, the array structure would be particularly susceptible to transient errors that could make entire rows or columns unpredictably inaccessible. In addition, this sort of massive redundancy would have a significant impact on the density of the memory.

There are several groups exploring nanotechnology answers to the memory problem. These nanotechnology devices and proposed array architectures will be discussed in chapter 2.

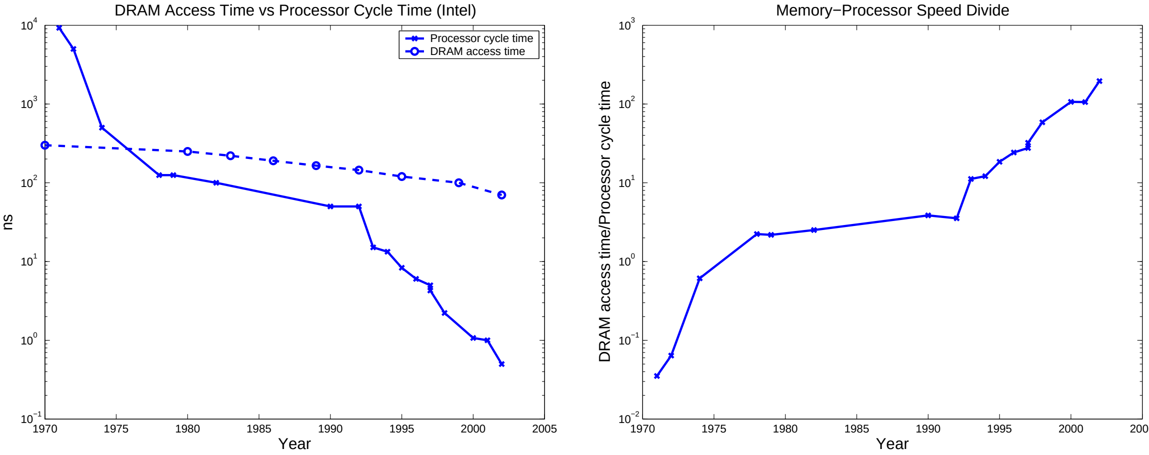


Figure 1.1. (a) The gap between DRAM access time and processor cycle times is wide and growing. (b) Moreover, the rate at which the gap is widening is increasing.

These barriers point to the need for a new kind of fundamental device and architecture, such as quantum-dot cellular automata (QCA). The device characteristics of QCA, which will be introduced below, are quite different from CMOS characteristics. This changes the cost landscape which in turn changes the look of efficient designs. The design framework presented in this thesis takes advantage of the characteristics of QCA as well as offering an option for alleviating or eliminating the von Neumann bottleneck.

1.2 An Architect's Introduction to Quantum-dot Cellular Automata

Quantum-dot cellular automata (QCA) is a novel alternative to the transistors, silicon, and CMOS paradigm. Rather than using charge movement, current, to propagate signals and perform operations, QCA uses devices as charge holders, using Coulombic repulsion of electrons as the primary computing force. A QCA cell consists of four quantum dots arranged in a square with two excess electrons that can occupy the dots. Because the electrons are repelled by each other, they naturally reside in opposite corners. As a result, the cell has two stable states. The first is an electron in the bottom left corner and the top right corner. A cell with this configuration has a polarization of +1 and represents logical "1". The second stable state is an electron in the top left corner and the bottom right corner, a polarization of -1 representing a logical 0 (figure 1.2). The electrons can tunnel between the quantum dots allowing them to change configurations.

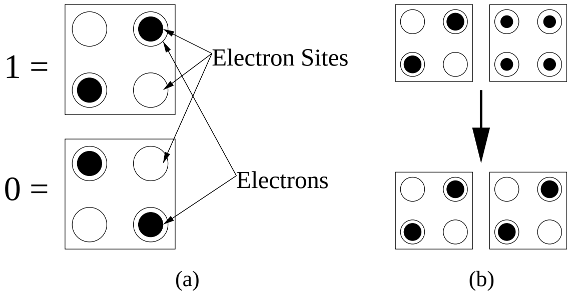


Figure 1.2. QCA Cell (a) Polarization and corresponding logic values, (b) Signal propagation in QCA. The cell on the left is polarized, the cell on the right is unpolarized. The cell on the right transitions to assume the polarization of the driving left cell.

Computation is performed by controlling the tunneling with a four phase "clock" signal (figure 1.3). Unlike CMOS circuits, the QCA clock is a fundamentally different phenomenon than the data. The clock raises and lowers the barriers between the dots, alternately prohibiting and allowing the electrons to tunnel between dots. The raising and lowering behavior of the clock signal is described by four phases called switch, hold, release, and relax. In the switch phase, the barriers begin low, allowing tunneling, and are raised to prohibit tunneling. In this phase, the cell transitions from having no value to having a definite value. The hold phase follows switch in which the barriers are maintained high, preserving the value assumed during switch. In the release phase, the barriers are falling, allowing the cell to go from a well-defined state to an undefined state in which the cell has no natural polarization. Finally, the relax phase maintains low barriers and no polarization.

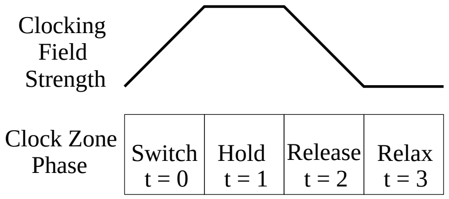


Figure 1.3. Propagation of clock signal in a single cell through time.

If QCA cells are lined up side by side and clocked appropriately, they act as a wire, propagating a signal down its length (figure 1.4a). Cells laid out in this side by side manner are called 90 degree cells. The alternative is 45 degree cells which are laid out corner to corner (figure 1.4b). In a 45 degree wire, the signal is inverted at each cell. If the first cell holds a “1”, the second cell will hold a “0”, followed by a “1” in the third cell, and so on.

QCA cells exist on a single plane. The two types of wires are able to crossover each other in this single plane without effecting the values being transmitted (figure

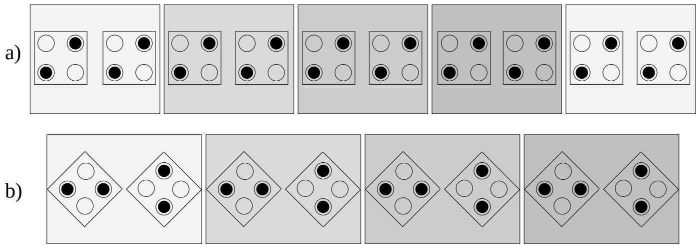


Figure 1.4. Shaded boxes indicate clocking zones. a) 90 degree cells forming a "wire". b) 45 degree cells forming a wire.

1.5). This makes complex circuits possible.

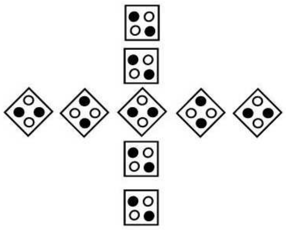


Figure 1.5. Wire crossover.

The basic logic gate in QCA is the three input majority gate (figure 1.6a). Three input cells are arranged on the edges of a center "device cell." The output of the gate is on the fourth edge of the device cell. The input cells and the device cell share the same clock zone. Because of this and simple coulombic repulsion, the device cell assumes the value of the majority of the inputs. When this device cell is frozen in the hold phase, it drives the output cell which then proceeds as a normal QCA wire. It is notable that the majority gate is a natural, native device in QCA. It requires nothing more than the QCA cells and clocking already introduced. This majority gate can be converted to either an AND gate or an OR gate by fixing one of the inputs to be permanently "0" (figure 1.6c) or "1" (figure 1.6d) respectively.

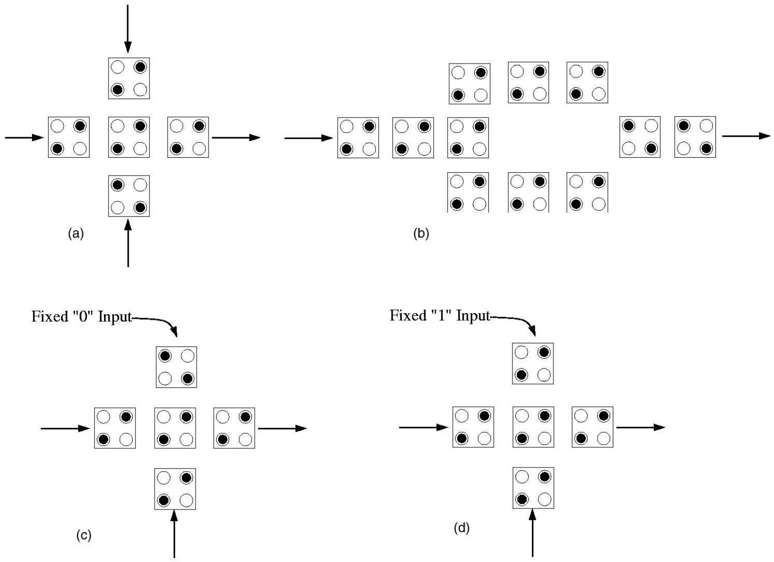


Figure 1.6. a) Three-input Majority Gate, b) Inverter, c) AND gate d) OR gate

An inverter is needed for logical completeness, and is formed by taking advantage of the 45 degree interaction (figure 1.6b).

In addition to these basic gates, the design landscape also includes three other important features. The first is the inherent latching in wires. In essence, the wires are shift registers. This adds a new dimension to designing QCA circuits rather than CMOS circuits, allowing a designer to pipeline at a very fine level. Connected to this inherent latching and pipelining, the second feature of QCA is the close connection between layout and timing [47]. There is an upper and lower bound on the size of clocking zones. Distances and time, then, are very tightly coupled. Finally, bits in QCA designs are always in motion. The clock and the cells are made of different technologies. Perhaps in the future it may be feasible to have the circuit influence the operation of the clock, but for the designs presented in this thesis, it is assumed that once the clock starts running, it continues to operate independently of the circuit. This, too, changes the design decisions made.

1.3 Prior QCA Architecture Work

Computer engineering QCA research first focused on device basic logical devices and an adder as an example of a QCA circuit [58]. Niemier's work was the first look at the effect QCA has on architecture and system design. His initial work focused on the hand designing of a simple but complete processor in QCA much as the first Intel 8086 processor was designed [39][4]. In the course of this work, Niemier identified several key elements of circuit design in QCA including the connection between layout and timing [47], the potential of processing-in-wire and fine-grained pipelining [46] [43] [42] [41], and initial floorplanning for logic [39]. In addition, since the first molecular QCA circuits that will be fabricated will need to be regular structures, the design of implementable FPGAs was explored [45] [44]. Another key work explored the layout parameters and layout rules that will govern the layout of QCA circuits [40].

Niemier's work identified key issues in the design of QCA logic circuits and systems. The principles he identified apply to memory as well as logic, but memory has a different set of important requirements.

Research is also being pursued to build fault models for QCA circuits in order to build fault tolerant circuits and to build CAD tools to facilitate testing and design of circuits [16]. In addition, the first algorithm that addresses the circuit partitioning problem in QCA has been developed [6].

1.4 Note on Circuit Figures

The circuit figures in this work require a brief comment. Most of the QCA figures presented in this work are drawn using a tool called QCADesigner [60]. The shadings of the cells indicate their clocking zone. For instance, figure 1.7 illustrates two wires with data propagating to the left. In this example, each clocking zone has three cells per wire in it. The top wire is a 90° wire, while the bottom wire is a 45° wire. Each wire passes through five clocking zones. The first four correspond to switch, hold, release, and relax respectively. The last clocking zone begins to repeat the pattern and is in switch. It is important to note, also, that unlike the figures presented previously, the cells in these figures are merely place holders and do not represent the polarization of each cell.

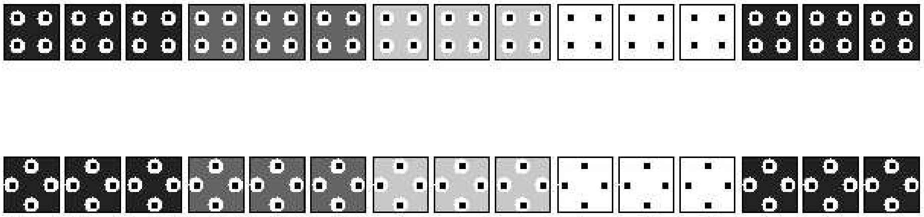


Figure 1.7. Example of 90° and 45° wires drawn with QCADesigner. The shading indicates clock zone. The top wire is a 90° with data moving to the left. The bottom wire is a 45° wire with data moving to the left.

1.5 The Real Device

QCA is very real. QCA cells have been fabricated and their operation experimentally verified [8] [48]. These QCA cells were constructed with metal dots on a micron scale and operate at 70 mK. As the size of the cell grows smaller, the operational temperature will rise [33]. A molecular implementation, then, would allow room temperature operation as well as offering significant potential density gains in circuits. Lieberman, *et al* have investigated several two dot-molecules such as the Creutz-Taube ion and mixed-valence ruthenium dimers. In addition, they have explored options for attaching these molecules to etched self-assembled monolayers [36]. Other groups at Notre Dame are investigating four-dot molecules[35] and alternate fabrication strategies such as DNA tiling.

In addition to the QCA cells, a functioning QCA circuit requires a clock signal and input/output capabilities. Lent, *et al* have designed an implementable clocking scheme in which buried metal wires are used to create the clocking field [26]. Bernstein, *et al* are investigating mechanisms for detecting the output of QCA circuits. The output of the metal-dot systems were detected using single electron transistor electrometers [8] [36].

Current estimates place fabrication of simple molecular circuits being possible within three to five years. More complex circuits and large scale fabrication will require more time, but are expected to be possible before the end of the roadmap is reached and nanoscale devices are required to meet density, speed, power and performance demands.

1.6 Introduction to the H Memory

This thesis presents a memory framework for QCA and several enhancements on the design. Some architectural work has been done designing memory [9] and processors [4]. However, this is the first effort in designing "native" QCA storage structures. Because of this, the proposed architecture is a departure from traditional SRAM and DRAM designs. These designs can be implemented in QCA, but the resulting designs are awkward and inefficient, providing only minimal gains over CMOS. The goal, then, was to design a highly dense memory native to QCA - to take advantage of properties of QCA in ways that today's memory designs take advantage of the properties of CMOS. The basic tool chest, as mentioned above, contains the majority gate, the inverter, the inherent latching and pipelining in QCA wires, the connection between layout and timing, and finally the constant motion of data.

The design that these tools lent themselves to is a serially accessed structure based on a binary tree arranged in a recursive H structure (figure 1.8). Memory macros store a word of data and contain the logic necessary to satisfy read and write requests. Router macros send memory requests toward the appropriate memory macro. In essence, accessing data becomes a routing problem. Data is stored at the leaves of the H-tree, and internal nodes are routers. In the basic framework, all memory accesses originate at the root of the memory and consist of two parcels of bits that travel on two parallel lines in lockstep with each other. One parcel contains the address, write enable, and data word. The other is responsible for signaling the presence of meaningful bits to the routers and leaves. This routing approach to accesses is natural to QCA and offers interesting architectural opportunities. In particular, it allows a simple way to incorporate logic into the memory structure

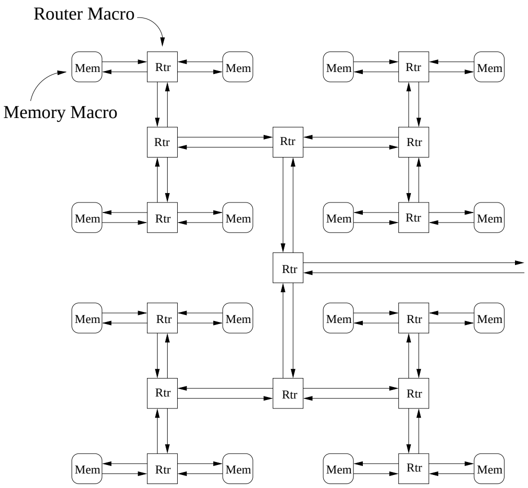


Figure 1.8. Basic H-memory Layout showing the organization of memory macros labeled “mem” and router macros labeled “rtr”. Accesses enter at the root on the right and travel through the internal router macros to a memory macro leaf, then back through the router macros to the root to exit the memory.

itself. Performing simple ALU functions in the routers or leaves can alleviate the von Neumann bottleneck, and fully incorporating the logic of the processing unit into the memory eliminates the von Neumann bottleneck altogether by eliminating the "central" of the central processing unit.

1.7 Original Contributions

My original contribution to this thesis includes the H-memory access method; the design, implementation, and analysis of the data loops and H-memory; the leaf enhancement design and analysis; the collision and execution time analyses for the bouncing threads enhancement; the simulator that allowed the exploration of the bouncing threads model in the presence of collisions; and finally, the discussion and analysis of the diagonal clocking wire layout.

1.8 Thesis Map

This chapter introduced the problem addressed here and the basics of the QCA device. Chapter 2 will discuss current storage structures including the different types of memories, silicon friendly architectures, and non-conventional memories. Chapter 3 will discuss the H-memory architecture. Chapter 4 will discuss a first cut implementation and its predicted performance. Chapter 5 will discuss two enhancements to the basic memory structure. The first is storing multiple words in each memory macro. The second is incorporating logic into the memory structure. Chapter 6 discusses an alternate clocking wire layout strategy and the corresponding effects on architecture. Finally, chapter 7 concludes and discusses some future work.

CHAPTER 2

Today's Memories

Traditional computer architecture has split its focus between logic and memory. Logic is responsible for computation, and memory is responsible for storing the state necessary for the desired computation. This state includes data with a wide range of persistence times, from short term data placed in buffers to permanent configuration information stored at fabrication time. Today's memories are based on arrays of memory cells. There are two main characteristics on which to classify these memories: volatility and access method. A third important classification subdivides the access method into storage method, either static or dynamic (figure 2.1). The latter half of this chapter discusses a few emerging nanotechnology proposals that may challenge contemporary commercial memories in the future.

The first axis on which to classify is volatility. Memory can be either volatile or non-volatile. Volatile memory is usually used for data that changes frequently such as main memory contents and usually allows fast read and write times. However, it needs to be connected to a power source and may need to be refreshed periodically. Non-volatile memory retains its data even after being removed from its power source. Hard drives and flash memory sticks are common examples of non-volatile memory.

The second axis on which to classify memories is the access method, consisting of read-only memory, random access memory, and content-addressable memory. Read-only memory (ROM) or read-mostly memory is generally non-volatile and

	Volatile		Non−Volatile
	Dynamic	Static	
Random Access	DRAM	SRAM	FeRAM
Content Addressable		CAM	
Read Only			EPROM EEPROM Flash

Figure 2.1. Memory Type Classification

used for data that is expected not to change, or to change rarely in the case of read-mostly memory. Random access memory (RAM) is a read/write memory, generally volatile, and used for data that changes often. Content-addressable memory (CAM), or associative memory, is a random access, read/write memory that allows access by data rather than by address. RAMs are presented an address and return the data associated with that address. CAMs are presented a data word and return the address with which that data is associated. RAM can be further subdivided into dynamic versus static storage. Dynamic RAM (DRAM) stores its data on a capacitor and requires a periodic refresh. Static RAM (SRAM) stores its data in a latch and avoids the need for a refresh.

The ideal memory would combine non-volatility, fast read times and fast write times. However, these three goals are often directly conflicting. For instance, non-volatility requires that data be hard to erase. Fast write times, on the other hand, require easily erased materials. Often times, easily writable materials lead to difficult to read materials since the read process must be very gentle relative to the write process. For the present, non-volatility is generally joined with read oriented applications, and volatility is tolerated by write oriented applications.

2.1 Array Memory

Each type of memory has a unique memory cell that does the actual bit storage, but most of these memories share a basic array architecture. The magnetic, optical and holographic disks are the exceptions. Memory cells are placed at the intersections of vertical and horizontal wires, or columns and rows. They are also known as wordlines and bit-lines, respectively. A bit is accessed when both its wordline and bitline are activated. In figure 2.2, a standard dynamic random access memory architecture is shown. The memory itself will be discussed below, but the architecture contains all the pieces common between the array memories. A particular bit in the array of bits, or cell matrix, is identified by an address. In the example, there are 10 address bits. Five bits identify the desired column (C1..C5), and five bits identify the desired row (R1..R5). Decoders on the column and row inputs translate the five bit partial addresses into one of 32 wires. When the address is presented to the address input buffers, the one bit data word at the intersection of the active row and column is selected to be operated upon (either read or written). This is a very simple example of an array architecture. Modern array memories contain multiple banks that allow larger data words to be used and to support faster data access.

2.2 Volatile Random Access Memory

Random access memories are generally volatile. This is generally appropriate since randomly accessed data tends to change often, requiring a quick and efficient write process. For instance, main memory and caches in general purpose computers commonly use dynamic RAM (DRAM) and static RAM (SRAM). DRAM is used where high density or low cost is important. SRAM is generally used where high speed is

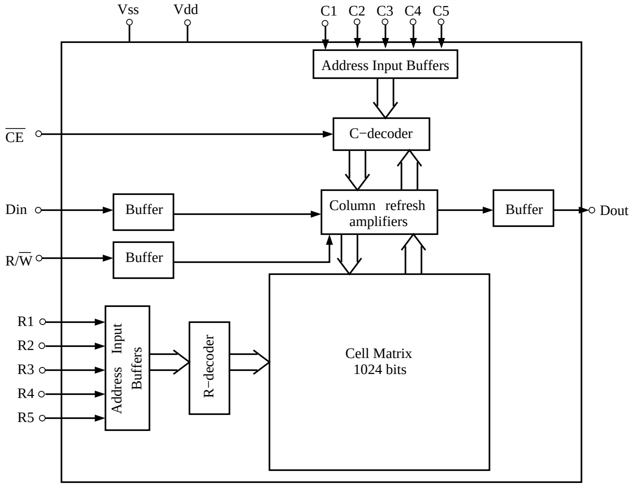


Figure 2.2. Basic array architecture for 1024x1 configuration, 10 address bits [30]

necessary.

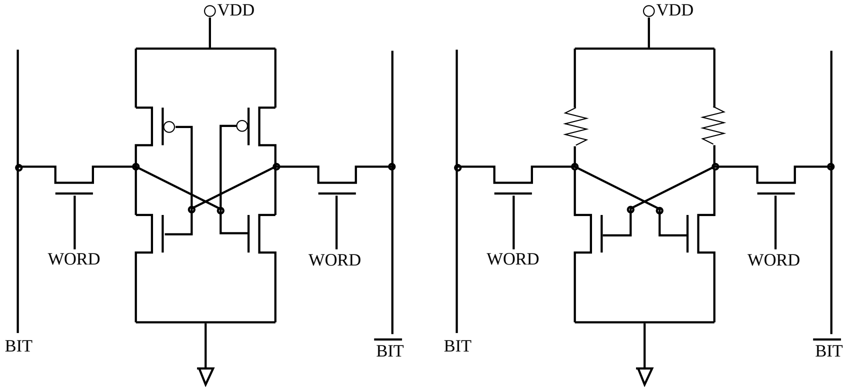


Figure 2.3. (a) 6 Transistor SRAM cell. (b) 4 Transistor SRAM cell

The static and dynamic options for memory cells generally trade off speed and area (figure 2.5). Static RAM (SRAM) stores data on latches and maintains its value as long as the power is on. Dynamic RAM (DRAM) stores the bit on a capacitor and requires the value to be refreshed periodically (figure 2.4). DRAM cells have a

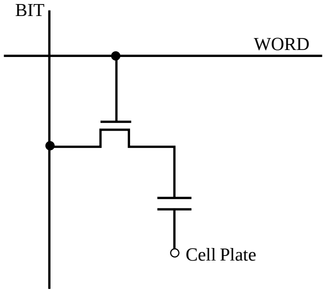


Figure 2.4. A one transistor, one capacitor DRAM cell

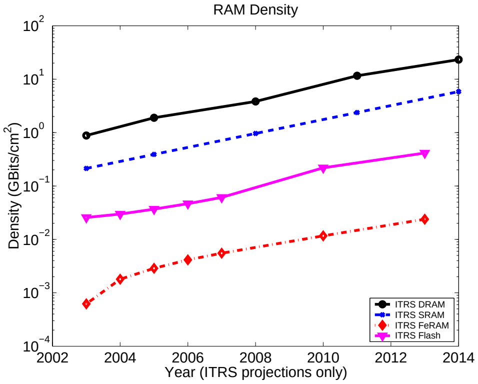


Figure 2.5. Density projections for SRAM and DRAM [28]

typical retention rate of 64 ms [28].

SRAM cells consist of a standard cross-coupled inverter latch and two additional transistors to control access to the data. The latch can be implemented either in CMOS with six transistors or in pseudo-NMOS with four transistors (figure 2.3). SRAM is between eight and sixteen times faster than DRAM since it does not lose time for data refreshing and for recovering from destructive reads. However, the SRAM cell requires four to eight times more area than the DRAM cell.

DRAM is generally used in applications such as main memory where latency can be tolerated for higher capacity. SRAM is used for applications where speed is essential, such as first-level caches.

2.3 Volatile Content-Addressable Memory

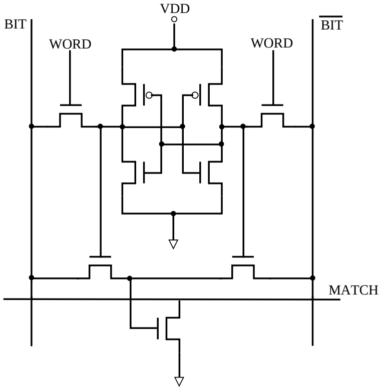


Figure 2.6. Content-addressable memory cell [50]

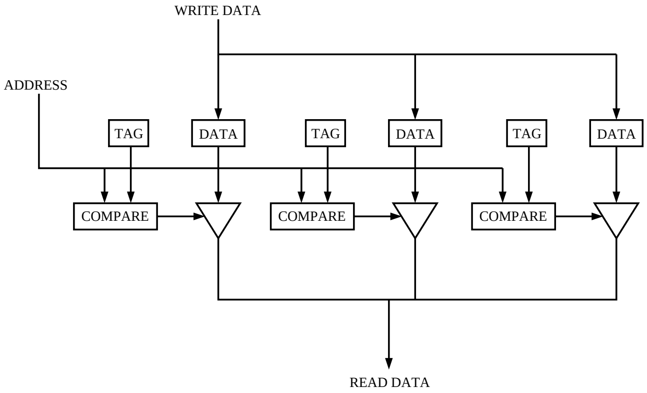


Figure 2.7. CAM architecture for a fast cache [51]

As opposed to standard RAM in which the memory is presented an address and the data is returned, CAMs are presented with partial data and either return the rest of the data or the address associated with that data. CAM cells can be made either with nine transistor cells (figure 2.6) or by adding a comparator to every address of a standard SRAM (figure 2.7). Modern manufacturers such as Motorola advertise CAM cells using as few as 4 transistors. The relatively large cell size limits the use of CAMs since they are relatively expensive per bit of storage. However, for applications that are search intensive and require fast access, such as a first-level cache, CAMs can be very effective. Stand alone CAMs are also used in tasks such as network routing and Virtual Path Identifier/Virtual Circuit Identifier translation in ATM switches.

2.4 Read-Only Memory

Read-only memories include memories such as EPROMs, EEPROMs, and CD-ROMs. EPROMs and EEPROMs are more properly called read-mostly memories since they can be reprogrammed a limited number of times, but these devices are focused toward applications in which the data is unlikely to change. These tend to be non-volatile devices since the data is stable over time and will be discussed in detail below.

2.5 Non-Volatile Memories

Non-volatile memories maintain their data in the absence of a power supply and can take many forms such as programmable read-only memories, modified RAM technologies, and disk technologies. In addition, there are flash memories and ferroelectric RAMs (FeRAMs). Programmable read-only memories tend to have expensive manufacturing costs compared to the standard volatile memories but have the advantage of non-volatility. Non-volatile memories made from modified RAM include shadow RAMs focused towards applications with a small number of set-up parameters and battery-backed SRAM (BRAM) when fast access and non-volatility are both important.

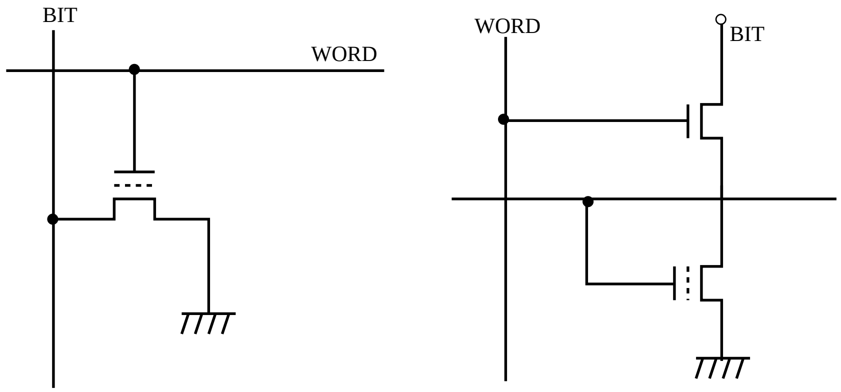


Figure 2.8. Erasable programmable ROM cell and electrically erasable programmable ROM cell [50]

The programmable read-only memory family includes erasable programmable read-only memory (EPROM) and electrically erasable programmable read-only memory (EEPROM) (figure 2.8). EPROMs are erasable, but the procedure requires the memory to be removed from the system and erased and reprogrammed by specialized equipment. This is appropriate for applications where the re-writing procedure is rarely needed. EEPROMs can be written a limited number of times, on the order of ten thousand, but it also allows in-system reprogramming in addition to non-volatility.

Magnetic disks are commonly used today. They are a type of writable, non-volatile, random access memory. They are not array memories. Rather, they consist of platters of magnetic material arranged in tracks and sectors. Tracks circle the platter while sectors split the disk into pie-piece shaped pieces. The data is stored by the magnetic polarization at a particular spot on the track. The platter rotates under a read/write head that can detect the magnetic field of each bit. The head is also responsible for writing a bit to be stored. Because magnetic disks use physically moving parts, they are slower and more prone to failure than semiconductor memories. However, they are cost effective for high capacity, latency tolerant applications. They are commonly used as hard drives.

Optical disks, too, are not array memories. They store data as areas with high or low reflectivity. Data is read and written using a low-power laser. Today's compact disks (CDs) and digital versatile discs (DVDs) are examples of optical disks. They have the potential for much higher densities than magnetic disks mainly due to the increased control over the read/write apparatus. The laser of an optical disk can be controlled more finely than the read/write head of a magnetic disk. However, today's optical disks are either read-only or can be written only a handful of times.

Holographic storage devices, rather than as an array, store data as holograms in photo-refractive crystals. This technology is not yet available commercially, but it has the potential for storage much denser than magnetic disks and an order of magnitude faster input and output. Several companies claim to be ready to release holographic storage devices in the next year. By making use of the properties of holographs, massively parallel reads and writes can be performed. In addition, multiple holograms could be superimposed by altering the angle of the access or the wavelength of light used.

Flash memory is a variation of EEPROM, an array memory. It is addressed at a block level rather than a byte level. The coarser granularity allows writes to be performed much more quickly than EEPROMs. Flash memory makes use of floating gate transistors which, if controlled precisely, can store two bits on one transistor. Standard flash memory stores a single bit per transistor, but the more sensitive two bits per transistor memory has twice the density of the standard flash memory. Flash memory is generally used for data that changes infrequently and for which non-volatility is essential such as basic input/output systems (BIOS) in personal computers, cell phones, digital cameras, and embedded controllers. Flash makes updating these systems simple and efficient. However, the tradeoff with the higher write speed is the lack of bit-level addressing, making flash an uncomfortable fit for RAM applications.

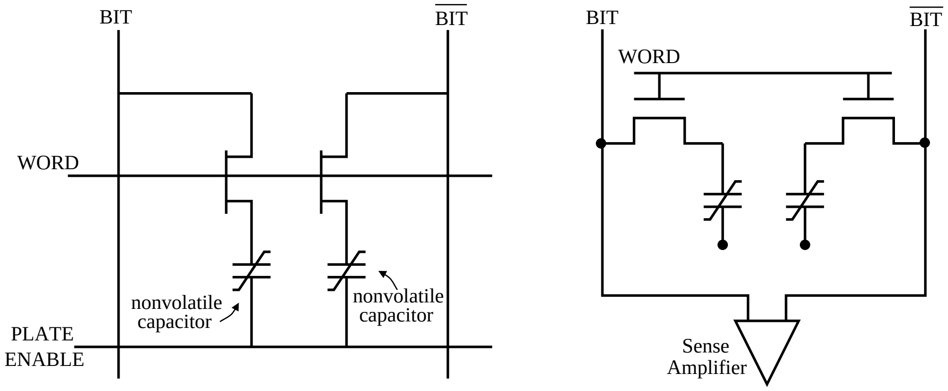


Figure 2.9. Two Schematics for two transistor, two capacitor non-volatile ferroelectric RAM cell, without and with sensing scheme. [52]

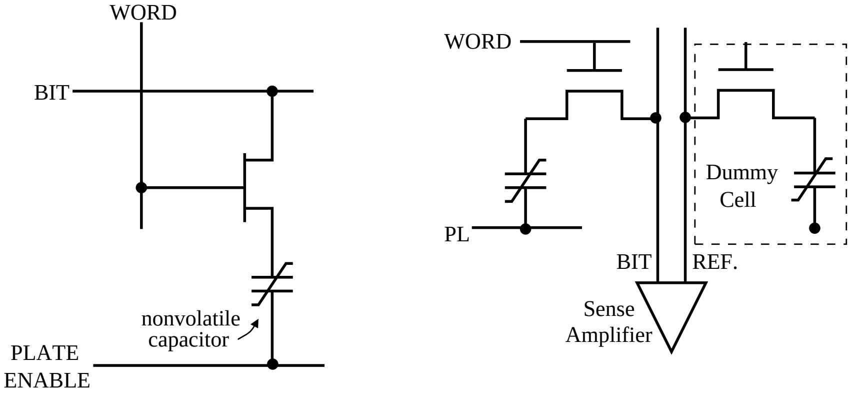


Figure 2.10. One transistor, one capacitor non-volatile ferroelectric RAM cells, without and with sensing scheme. [52]

Ferroelectric memories (FeRAMs) store data in a capacitor with a ferroelectric dielectric. Rather than storing data as free charge in a capacitor, the data is stored by the electrical polarization of crystals within the ferroelectric film [52]. The charge on the ferroelectric capacitor cannot be directly detected, only the change in polarization can be detected. There are two common cell configurations, a 2T-2C cell (figure 2.9) and a 1T-1C cell (figure 2.10). The 2T2C cell is more stable than the 1T1C cell, but it is not competitive with other memories in terms of area. The 1T1C FeRAM cell addresses this area difference, but there are several challenges to be overcome before the 1T1C cell becomes practical. For instance, the dummy reference cell used for reading will build up remnant potential on its capacitor after reading a logical “1”, changing the reference point for the next read cycle. If the cost and density of FeRAM become competitive with DRAM and flash, it will present a powerful random access, non-volatile memory. The area of a 2T2C FeRAM cell is comparable to that of SRAM, but it is far larger than DRAM or flash. The write process of FeRAM is substantially faster than the write of flash memory, making the development of more area efficient FeRAM cells very attractive for write oriented non-volatile applications.

Battery backed SRAM (BRAM) is a standard volatile SRAM with additional circuitry and packaging such that when external power is removed, the device senses this and switches to battery power to maintain the stored data. While not truly non-volatile since the battery has a finite life, BRAM combines the fast, random access of SRAM with pseudo non-volatility.

2.6 Nanotechnology and Nanomemory

The main thrust of nearly all architecture research at the nanoscale has focused on variations of the array architecture familiar on current RAM designs. The standard approach involves inserting the nanoscale device at the intersections of a wire array. The nanoscale devices generally proposed to be used in this way include either two- and three-terminal devices that act much as today’s micro-scale diodes and transistors, or nano-mechanical devices such as rotaxanes and the buckyballs in the shuttle memory proposals. Other proposals involve the physical presence or absence of molecules and more complex reading/writing procedures as in the silicon atom lattice.

2.6.1 Two and Three Terminal Device Memories

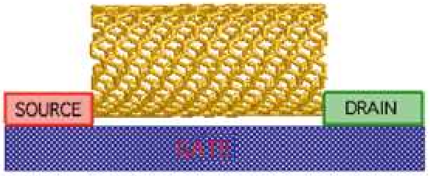


Figure 2.11. Schematic for a possible nanotube field effect transistor. [2]

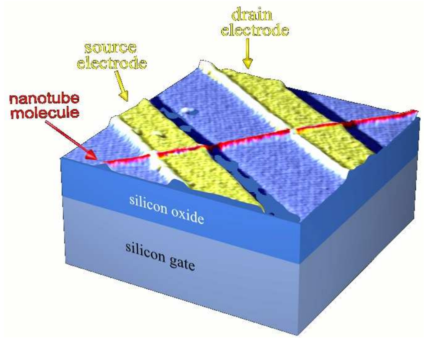


Figure 2.12. Image of an implementable nanotube FET [3]

There are several proposals to make use of two and three terminal nano-scale devices to implement standard RAM architectures at the nano-scale such as Goldstein and Rosewater [23], Dehon [14], Yano, et.al. [63] and Nackashi and Franzon [38], among others. Goldstein and DeHon use RAM structures to build programmable

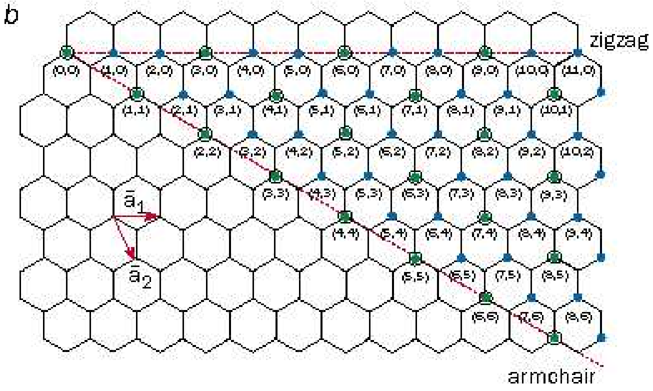


Figure 2.13. The chiral vector of nanotubes is defined by na_1 + ma_2, where n and m are integers. Armchair nanotubes have $n = m$ and chiral angle of 30°. Zigzag nanotubes have either n or m equal to zero and chiral angle of 0°. All other nanotubes are considered chiral. [15]

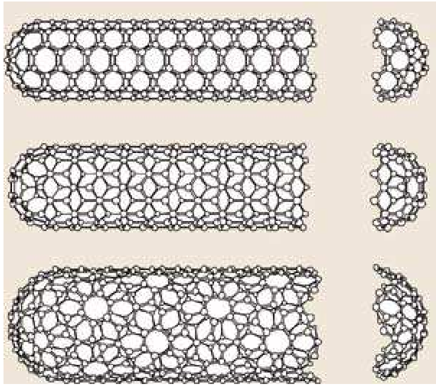


Figure 2.14. Three single-walled carbon nanotubes with different chiralities. Pictured here is a (5,5) armchair nanotube (top), a (9,0) zigzag nanotube (middle) and a (10,5) chiral nanotube. [15]

logic array (PLA) type general-purpose computing platforms. Yano and Nackashi focus on using a standard RAM architecture to test their devices.

Several architectures are being proposed for a three terminal field effect transistor making use of semiconducting carbon nanotubes (figures 2.11,2.12). Carbon nanotubes can be thought of as narrow sheets of pure carbon rolled into a cylinder (figure 2.14). They can be grown as either single-walled (SWNT) or multi-walled (MWNT). MWNTs have multiple telescoped nanotubes. SWNTs consist of a single nanotube. Because of the unique chemical composition of nanutubes (pure carbon), they have many interesting characteristics. For instance, depending on how the hexagonal carbon lattice is oriented, the tubes can be either metallic (conducting) or semiconducting. Also, under certain circumstances, carbon nanotubes can become ballistic conductors. This means that the electrons that pass through the tube will not be scattered, significantly speeding up the passage of the electron.

The conducting behavior of the nanotube is determined by the "roll angle", called the chirality of the nanotube (figure 2.13). The chirality is defined by the chiral vector. The chiral vector is calculated by na_1 + ma_2, where a_1 and a_2 are defined as in figure 2.13, and n and m are integers. There are three chiralities: armchair, zigzag, and chiral. An armchair type nanotube has $n = m$ and a chiral angle of 30°. A zigzag nanotube has either n or m equal to 0 and a chiral angle of 0°. All other nanotububes are considered chiral. A nanotube will be metallic, or conducting, when $n - m = 3q$, where q is an integer. All armchair nanotubes and approximately 1/3 of possible zigzag nanotubes are metallic [15]. All chiral nanotubes and the remaining zigzag nanotubes are semiconducting. All of these types of nanotubes are being considered for use in different devices.

Lieber's group proposed a non-volatile carbon nanotube switch that could be used for a non-volatile random access memory. A non-volatile switch can be formed by suspending a nanotube across either another nanotube or a nanowire. The upper nanotube is in a stable state when it is undeformed, this corresponds to the "off" position of the switch. The nanotube can be deformed by running the appropriate attractive currents through the upper and lower levels. When the upper nanotube is brought close enough to the lower nanotube or wire, van der Waals forces create a second stable state that maintains the deformation and corresponds to the "on" position of the switch since there is very little resistance between the deformed nanotube and the lower nanotube or wire (figure 2.15).

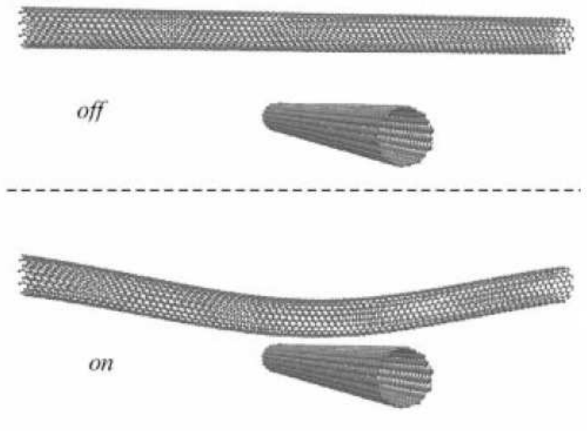


Figure 2.15. Nonvolatile nanotube switch in off and on positions [54]

Dehon has also taken a reconfigurable approach to architecture at the nanoscale with "nanoarrays." Several devices could be used as the functional devices in the nanoarrays. The first possible device is the nanotube switch proposed by Lieber's group discussed above. The second possible device discussed is a nanowire FET which can be constructed by appropriately doping semiconductor nanowires [27]. The third device proposed for use in the nanoarrays are the non-volatile rotaxane and catenane switches proposed by Heath and Stoddard and discussed below. The architecture proposed by Dehon makes use of crossed arrays of wires with the devices at the intersections. The crossed arrays can then be used to create memory cores, programmable logic arrays (PLA), and crossbars to create a reconfigurable computing device. A simple architecture would consist of all programmable FET arrays. More complex organizations could alternate non-programmable FET logic arrays and programmable diode arrays to create a programmed array logic (PAL) like architecture. At the core of PLA and PAL architectures is a crossbar archticture. At the microscale, fully populated crossbars are infeasible as the switches quickly dominate the area. At the nanoscale, however, fully populated crossbars are feasible since the switches no longer dominate the area. This greater population improves the defect tolerance of the circuit, providing more rerouting and reprogramming options. A final concern of the nanoarray architecture is to support connections to the microscale world. The microscale wiring can be seen in figures 2.16 and 2.17. The number of microscale wires needed scales logarithmically with the size of the nanoarrays. For large nanoarrays, then, the microscale wires become a thin frame around the large nanoarray core.

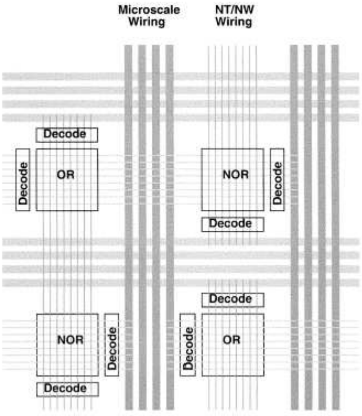


Figure 2.16. Overall assembly of Functional Nanoarrays [14]

Goldstein and Rosewater proposed a reconfigurable architecture based on chemically assembled electronic nanotechnology (CAEN) using molecular resonant tunneling diodes (RTDs), a two terminal device. RTDs are not restricted to the nanoscale, and are used today in certain applications. However, at the nanoscale, they allow architects to draw on previous work with resistor-diode logic. For instance, figure

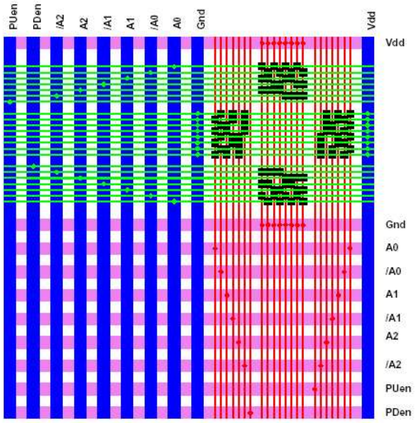


Figure 2.17. 8x8 nanoscale array bracketed by decoders and connection to microscale wires. Note that the number of microscale wires grows logarithmically in array width. For large nanoarrays, the microscale wiring becomes a thin frame around a large nanoscale core. [14]

2.18 shows the schematic for an AND gate using four diodes out of a block of nine diodes. At a very high level, the resulting architecture is similar to that presented by Dehon, consisting of a reconfigurable crossbar array. The basic building block is a "nanoBlock" (figure 2.19). A nanoBlock consists of a 2-D grid of reconfigurable diodes. Inputs enter the block from the north and west. Outputs exit the block on the south and east. The blocks can be abutted to build more complex structures. This architecture also makes an explicit connection to the microscale world, relying on a CMOS layer to provide power and ground connections.

The work by Yano, et al. [63] is not strictly nanoscale, but it demonstrates the use of a nanoscale single-electron memory device wired into a microscale circuit. The memory device is based on the "single-electron box" (figure 2.20). The single-electron box uses the Coulomb blockade effect to precisely control the number of electrons on the central dot. The number of electrons at the dot determines the value of the bit stored there. These devices have been fabricated using thin (below

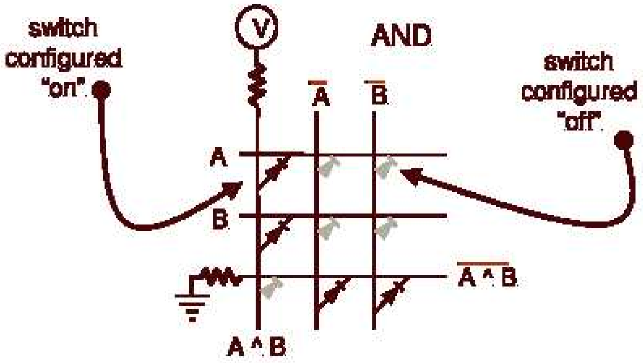


Figure 2.18. CAEN implementation of a Two-input AND gate. [23]

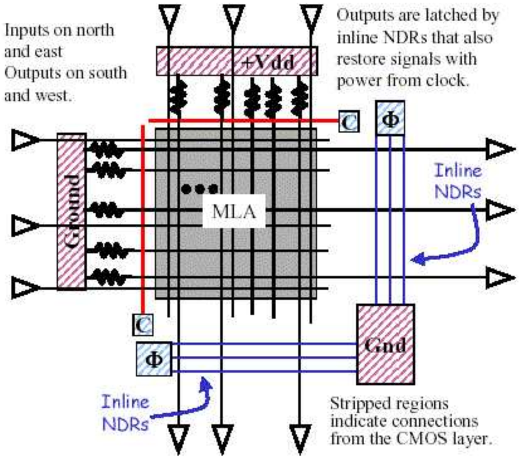

Inputs on north and east
Outputs on south and west.
+Vdd
Outputs are latched by inline NDRs that also restore signals with power from clock.
C
Φ
Inline NDRs
Ground
MLA
C
Φ
Gnd
Inline NDRs
Stripped regions indicate connections from the CMOS layer.

Figure 2.19. A schematic of a nanoBlock.[23]

5 nm) films of nanocrystalline silicon. The schematic of the device and a scanning electron microscope (SEM) picture of the actual device can be seen in figure 2.21. They extended the single device into a ladder-shaped memory cell array (figure 2.22). The 8x8 bit memory displayed in the figure is at the microscale rather than nanoscale, but it demonstrates a possible path toward nanoscale memory.

These proposals all have potential for usable nanoscale devices, but they all are

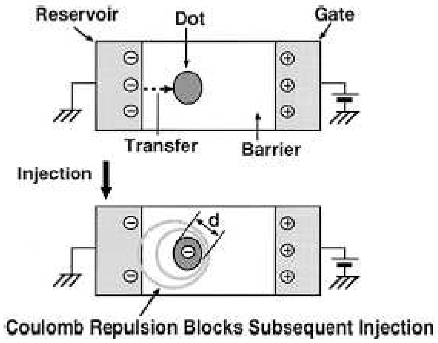


Figure 2.20. Schematic view of single-electron box. [63]

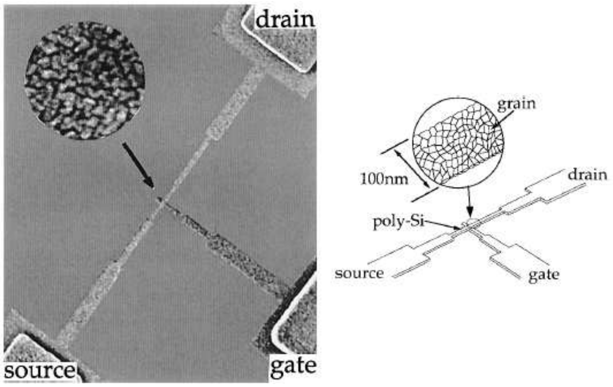


Figure 2.21. Memory device formed by sandwiched nanosilicon particles, SEM micrograph and schematic. [63]

based on the transistor paradigm. This means they are able to leverage decades of research and architecture work, but they also bring with them the weaknesses of the paradigm that the nano-scale world is not able to tolerate as well as the microscale world. The most significant of these weaknesses may be tolerance of fabrication defects and transient faults. In the world of CMOS, a single faulty wire or transistor generally renders the entire module non-functional. There are some well known exceptions. For instance, RAM chips have extra rows designed into them

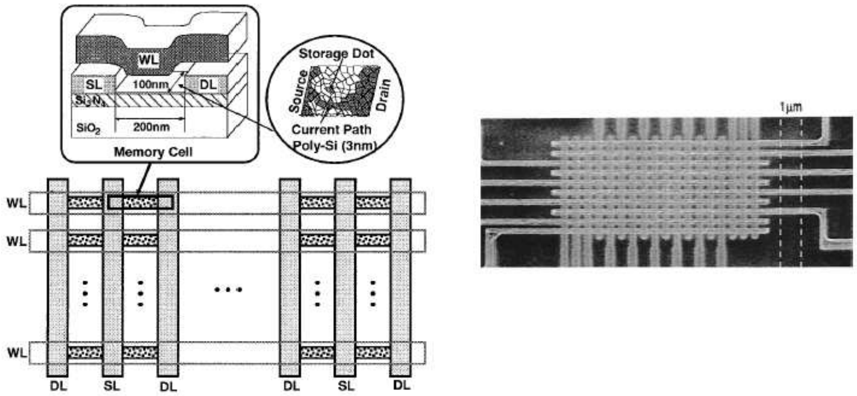


Figure 2.22. Overall potential memory architecture, schematic and SEM of micro-scale prototype.[63]

to allow chips to tolerate a few faults. More significantly, the world of reconfigurable computing with PLAs and FPGAs allow faults to be routed around when they are discovered. Perhaps the Teramac machine is the best example of this [25]. It is well accepted that nanotechnology fabrication methods will be much more error-prone than today's microscale fabrication processes, and once fabricated, the devices will be more susceptible to environmental hazards. The solution to these greater error rates has been massive redundancy along the lines of the Teramac architecture. However, it is unclear what level of reconfigurability and redundancy is necessary to maintain a working nano-scale computing platform and how responsive these reconfigurable architectures will need to be to tolerate transient faults.

2.6.2 Nano-mechanical Memory

The proposals discussed below have two key features in common. First, both of these architectures are based on an array structure and suffer from a common pitch-match problem. Since all the wires in the array need an interface to the micro-scale world, there is a danger that the micro-scale interface will negate the area savings gained by the nanoscale devices. While there are many well known strategies for minimizing the number of "pins" that lead off the nanoscale module such as multiplexing addresses and sharing pins by buffering inputs, this pitch-match problem persists as a significant issue. The second feature is that both proposals are only relatively non-volatile. How long the devices will maintain their value before spontaneously switching depends on the specific implementation, but it is not necessary to maintain a connection to a supply voltage to maintain the stored values.

Shuttle-memory

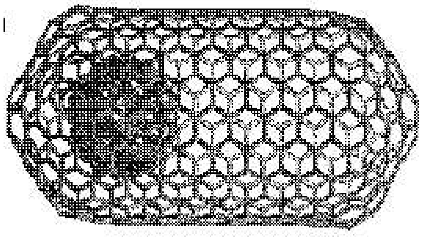


Figure 2.23. Structural model for a bucky-shuttle memory element in the logical "0" position. [32]

The buckyball-shuttle memory device was proposed by Kwon, et al in 1999 [32]. The memory device consists of a C_{60} buckyball "shuttle" contained in a C_{240} carbon nanotube "capsule" (figure 2.23). Once the buckyball is on one side of the nanotube, van der Waals forces will keep it at that end (figure 2.24). A bit is represented by which side of the capsule the shuttle is physically on. By attaching wires to the ends of the nanotube, the device can be written, and a destructive read can be performed similar to traditional DRAM read operations. A third wire attached to the center of the nanotube would allow a non-destructive read to be performed, but this third

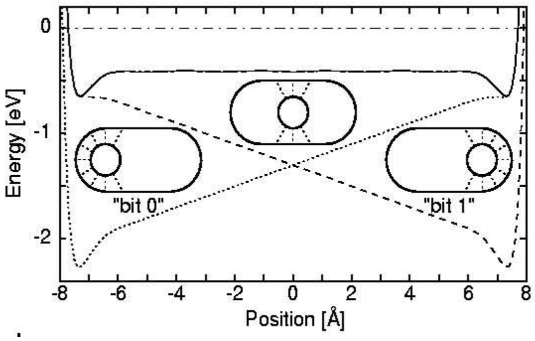


Figure 2.24. Potential energy of a particular potential implementation of the bucky-shuttle memory as a function of the shuttle's position compared to the capsule. [32]

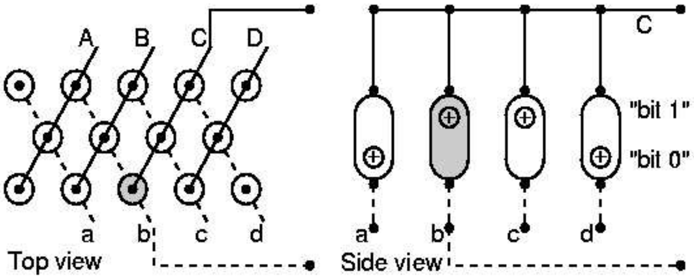


Figure 2.25. Schematic of high density memory board. [32]

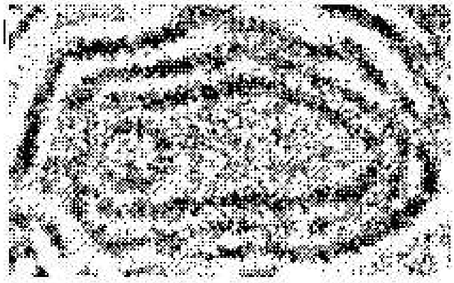


Figure 2.26. Transmission microscope image of multi-walled carbon nanotube that could be used as shuttle memory element.[32]

wire is not conducive to the scaled implementation mentioned below, presents a significant fabrication challenge to the nanoscale implementation, and will mitigate the area improvements gained by moving to the nanoscale.

One of the architectural goals in organizing this shuttle device was to present a realizable memory in the near term that could be scaled down to take advantage of fabrication advances [11]. The architecture consists of an array of vertical and horizontal wires with one or more shuttle devices at each intersection (figure 2.25). In the near term, for instance, 70 nm wide wires could be used in the array with approximately one thousand devices residing at each intersection. In this architecture, each intersection represents one bit. Ultimately, one can imagine the array being constructed from nanowires that require only a single device per intersection, and so per bit.

Using the C_{60} shuttle and the C_{240} capsule, each device is approximately 1.4 nm in diameter, and 2.0 nm in length (for instance, figure 2.26). Since a write involves physically moving the shuttle from one end of the capsule to the other, the device requires time to settle into its new state. The authors estimate this will be approximately 20 picoseconds, and an ideal memory switching and access rate of 10 GHz.

In addition to the pitch-match problem faced to interface to the micro-scale world, the density of the memory will be limited by the wire pitch of the array and potentially by the devices themselves as the heat generated by the collisions between shuttle and capsule becomes significant.

Rotaxanes and Catenanes

Another nano-mechanical storage strategy makes use of rotaxanes and catenanes. Rotaxanes are molecules consisting of a dumbbell shaped component and a ring

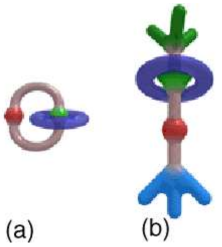


Figure 2.27. Sample nanomechanical memory devices. a) Interlocking ring catenane molecule. b) Dumbbell shaped rotaxane molecule. [1]

trapped on the dumbbell (figure 2.27b). Catenanes are molecules consisting of two separate rings looped through each other (figure 2.27a)[49]. The molecule naturally stores a binary value in much the same way as the shuttle-device discussed above. The physical position of the ring at either end of the dumbbell represents the two distinct values. The architecture used in exploring the fabrication of these devices is a crossed wire array with devices placed at the intersections of the wires [13]. The study described here uses 40 nm wide wires and 1100 molecules at each intersection. When the ring switches from one end of the dumbbell to the other, the electrical properties of the molecule change substantially, with the on/off resistance ratios ranging from 2 to 10^4 for different devices. It is important to note that the individual devices were being tested, as opposed a complete memory. However, this is a very promising result, demonstrating the potential of this technology. There are significant challenges to be overcome, though. For instance, the apparently short lifetime of the molecules as useful devices could cause problems. The on/off resistance ratio decreased after 40 cycles, approaching 1 after at most several hundred cycles.

Architectures for this type of molecule are being explored using them both as two terminal and three terminal devices (figure 2.28). The intersection of the hexagonal

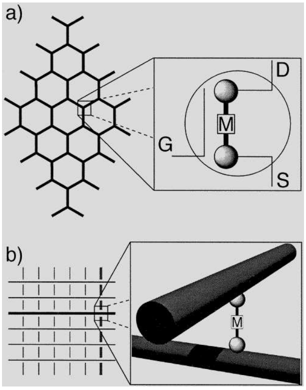


Figure 2.28. Possible tilable device architecures. (a) Three terminal device in a hexagonal lattice, where G signifies the gate, S the source, and D the drain. M represents a molecule. (b) A two terminal device used in a cross-bar architecture. [49]

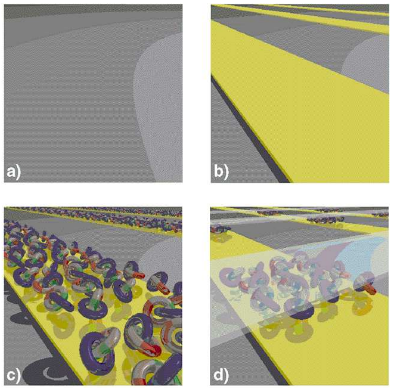


Figure 2.29. Fabrication of two-terminal molecular (catenane) switch. (a) Smooth silicon substrate. (b) Parallel etched polysilicon wires. (c) Deposition of catenane monolayer. (d) Second layer of wires perpendicular to the original wires, formed by condensing titanium vapor through a shadow mask. [49]

lattice creates a three-terminal FET device while the square lattice, or cross-bar architecture, leads to a two-terminal device architecture as discussed above. The hexagonal lattice presents a challenge in addressing the individual transistors, but there are several possibilities for overcoming this challenge. The hexagonal lattice lends itself more to logic devices than memory. The two-terminal device has been manufactured by Heath's group at Caltech (figure 2.29). Beginning with a specially prepared, extremely smooth silicon substrate, a series of parallel polysilicon wires are etched onto the substrate photolithographically. Next, the catenane monolayer is deposited, followed by the deposition of wires perpendicular to the polysilicon wires are formed by condensing titanium vapor through a shadow mask. The molecules at the intersection of the two wires are trapped, forming the two terminal devices. Because the position of the ring alters the conductivity of the molecule, a molecular switch can be demonstrated.

2.6.3 Silicon Atom Lattice

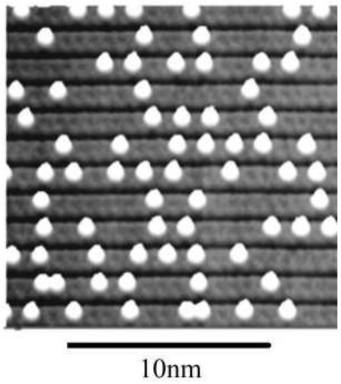


Figure 2.30. Silicon atomic lattice. Horizontal lines are gold tracks exactly five atoms wide (1.7 nm). White circles are a single silicon atom representing a bit. Along the tracks, the silicon atoms are seperated by 4 atom-widths. [7]

The proposal by Bennewitz, et al [7] departs from the wire arrays discussed above and pursues very high density at the cost of expensive read/write methods. Specifically, bits are represented as the presence or absence of a single silicon atom in a unit cell of 5x4 atoms (figure 2.30). Rather than the RAM-like architectures discussed above, this is more like a read-only or read-mostly memory. The memory

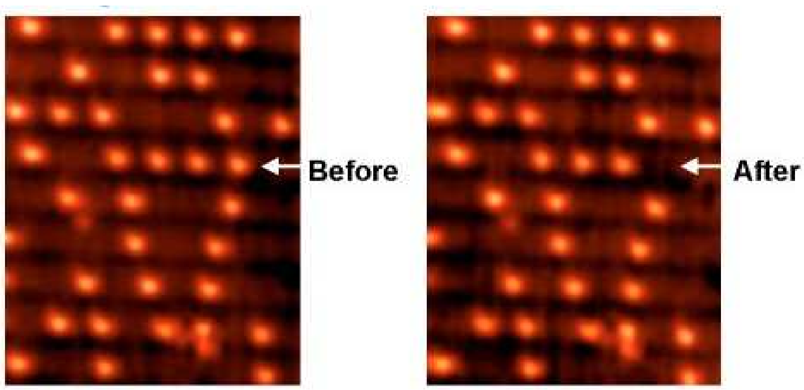


Figure 2.31. Writing a “zero” involves physically removing the cluster of silicon atoms using a scanning tunneling microscope. [7]

consists of lattice with self-assembled gold tracks and a pitch between tracks of exactly 5 silicon-atom-widths (1.7 nm). Within the track, bits are separated by 4 silicon-atom-widths. This is necessary to keep the stored bits from interacting with each other. An extra silicon atom is then deposited in each cell, effectively writing a logical “one” to each memory cell. Using a scanning tunneling microscope (STM), these excess silicon atoms can be selectively removed, writing a logical “zero” to these cells. It is also possible to return a silicon atom to a cell and rewrite a one, but this is an error-prone process using the STM.

This proposal is clearly neither convenient nor easily interfaced to conventional logic, but it demonstrates just how densely data can be stored. This proposal leads to densities on the order of 250 Tbit/in^2. The writing process is very slow and error prone, but the read-out rate estimated matches that of today’s magnetic hard disks for “low” densities (1-100 Gbit/in^2), and drops to 100 bits/s for high densities near 10^5 Gbit/in^2.

2.6.4 Other Non-Silicon Computing Proposals

To round out the picture of the nano or non-silicon world, there are two additional approaches that need to be discussed. They are true quantum computing and *in vivo* computing. Neither of these approaches has density as its primary goal, but they represent the other major members of the community.

Quantum Computing

The advantage of true quantum computing is a speed up in execution time rather than in area. Today's computers operate on bits that can be in one state at a time. n bits can represent one of 2^n states. Quantum computation is performed on qubits which represent superpositions of all possible states. n qubits represent 2^n states simultaneously. However, while qubits can represent multiple states simultaneously, only one state can be read out at the end of computation.

Regardless, quantum computation has the potential for exponential speedups over classical computation. Designing algorithms to exploit the power of quantum computers is a very active research area. Several algorithms have already been identified, the best known of which are Shor's factoring algorithm [56] and Grover's sorting algorithm [24].

There are several strategies being explored for implementing quantum computers. The primary approaches include ion traps, nuclear magnetic resonance (NMR), quantum dots, and optics. The ion trap approach uses electric and magentic fields to trap individual ions in small spaces. The ions themselves can then be used as qubits and manipulated by lasers. NMR quantum computers use the magnetic moment of nuclei as the qubits. The largest quantum computation executed to date used the NMR approach to factor the number 15 using seven qubits [59]. Quantum dots can also be used as qubits by creating a dot in which an electron can be confined at discrete energy levels. The electron can then be manipulated by focused lasers that control the electron's energy level. There are two general classes of optical quantum computation approaches both of which use photons as qubits. The nonlinear optical approach forces photons to interact with eachother in nonlinear materials. Linear optical approaches use linear optical devices such as mirrors and beam splitters to move photons and then use selective measurements to manipulate the value of the qubits.

Quantum architectures have not had an impact on this work because the bits being considered here are classical. In addition, memory and storage is very difficult for quantum computers. Maintaining a state without letting it interact with the environment is one of the major challenges faced by quantum computing researchers.

In vivo

In vivo computation involves designing a plasmid to be incorporated into a bacterial cell. Plasmids are genetic material held outside the cell nucleus. During cell division, plasmids are replicated and copies are given to both cells. It is easy to imagine two types of memory in these systems. The first is a nonvolatile instruction memory in the form of the DNA of the plasmid. This is a potentially highly dense form of static storage, using only 32 atoms per bit [7]. The I/O for such storage is difficult, though. Once the plasmid is made and inserted into the cell, it is difficult to alter it. One can imagine using this memory as a ROM, or pursuing more exotic write techniques such as designing a virus that would cut out pieces of the plasmid and replace it with the new DNA data.

A second technique is to use the building blocks demonstrated by Elowitz's genetic regulatory network oscillator [17] and Weiss's genetic building blocks [62] in which signals are transmitted via gene expression to construct a dynamic memory.

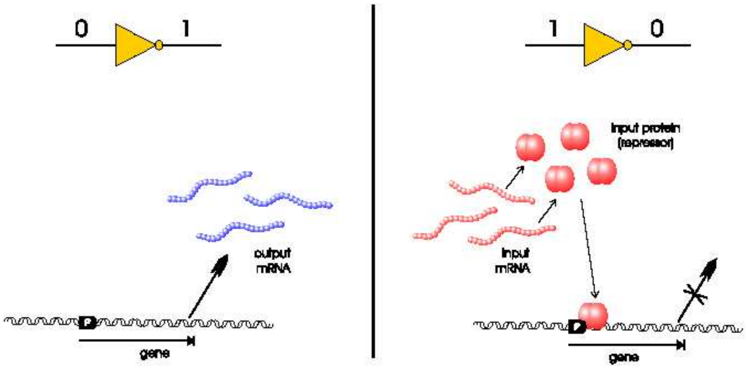


Figure 2.32. Simplified genetic inverter. On the left, the input is a logical “0”, allowing the gene to be expressed, producing mRNA output (a logical “1”). On the right, a mRNA is presented to the system (logical “1”) which represses the production of the output mRNA (logical “0”). [62]

For instance, figure 2.32 shows how a simple inverter would operate. This is a simplified view of the operation, but the idea is that when the gene which is the input mRNA is not present, the output mRNA will be produced, expressing the gene. This corresponds to a logical “0” input and an output of logical “1”. On the other hand, when the input mRNA is present, it will block the transcription, preventing the expression of the gene. This second case corresponds to an input of a logical “1” and an output of a logical “0”. One can imagine this approach being used to implement a volatile, dynamic biological memory with a simpler read/write process than is needed in the case of physically altering plasmids. However, the number of bits that could be stored in the system is potentially limited due to the number of protein combinations that could be used concurrently without interacting with either each other or other proteins needed to maintain the life of the cell.

2.7 Summary

The characteristics of the ideal memory have been identified as being “low cost, high performance, high density, with low power dissipation, random access, non-volatile, easy to test, highly reliable, and standardized throughout the industry” [50]. Each of the memories and technologies discussed above addresses at least one of the criteria set forth by Prince, but no nanotechnology has conquered them all. Active research continues to develop the technology and architecture to meet the challenge. If chemical self-assembly techniques can be harnessed, the world of nanotechnology should be able to present very low-cost, high-density memories. Low power will be essential for these nanoscale proposals since they will be unable to sustain reliable storage in the face of the high temperatures common in today’s commodity devices. Several of the proposals mentioned make use of non-volatile molecular or crystalline switches, addressing the sixth characteristic mentioned. Random access, testability, and reliability will all be issues to be solved at the architectural level. Reliability will be a particular challenge since the devices themselves will most likely be error prone and susceptible to environmental interference. While there is not a clear proposal at the nanolevel that will exhibit all of the characteristics in the ideal memory, the nanoscale proposals suggest the opportunity for great strides toward such an ideal memory. The nanomemorries discussed and end of the roadmap CMOS DRAM are compared and summarized in Table 2.1.

Table 2.1

Potential Memory Density of Proposed Technologies

Technology	Density	Comments
CNT switch	$567bits/\mu m^2$†	nonvolatile
Nano FET	$567bits/\mu m^2$†	leakage current
Catenanes, rotaxanes	$567bits/\mu m^2$†	nonvolatile, molecule short lived, after 40 cycles, on/off resistance ratio begins to decrease
Shuttle	$567bits/\mu m^2$†	nonvolatile, heat from shuttle movement potentially limits density
CAEN	$567bits/\mu m^2$†	crossbar arch with pitch match problem. Leverages past work on resistor diode logic
Nanoarrays	$567bits/\mu m^2$†	crossbar PLA/PAL, pitch match problem, unknown redundancy requirements
Nanosilicon Particles	$567bits/\mu m^2$†	implemented at microscale, but shows promise for nano implementation. pitch match problem.
Silicon Atom Lattice	$4 * 10^5 bits/\mu m^2$	dense, nonvolatile, slow and error prone write process, read time on par with hard disks, requires STM
Quantum Computing	N/A	includes ion trap, NMR, optical implementations, strength is execution time rather than density
In Vivo Computing	32 atoms/bit, or $2.7 * 10^5 bits/\mu m^2$	use either plasmid, nuclear DNA, or gene expression levels, uses live cells, complicated and limited protein interactions
CMOS DRAM [29]	$625bits/\mu m^2$	Using the red brick end of the roadmap DRAM cell size only. Does not include any limiting factors such as wire pitch or addressing requirements.

† Limited by wire pitch, assumes end of the roadmap (2018) metal 1 pitch of 42 nm [28].

CHAPTER 3

H-Memory Architecture

The H-memory architecture is an alternative to the standard 2D array memory organization. The 2D array structure can be translated into QCA, but it does not match the device strengths. The H-memory is a highly dense memory design native to QCA that takes advantage of the device. As discussed in chapter 1, the basic tool chest for QCA includes the majority gate, the inverter, the inherent latching and pipelining in QCA wires, the connection between layout and timing, and finally the constant motion of data. The memory design that these tools lend themselves to is a serially accessed structure based on a binary tree arranged in a recursive H structure (figure 3.1). Data is stored at the leaves of the H-tree, and internal nodes are routers. Memory macros store a word of data and contain the logic necessary to satisfy read and write requests. Router macros send memory requests toward the appropriate memory macro. In essence, accessing data becomes a routing problem. The final piece of the architecture is a simple return path that continues the pipelining on the outward path of the memory.

The memory attains its greatest improvement over standard CMOS RAM when memory accesses are entering the memory as fast as possible, i.e. as soon as one access fully enters the memory, another access is ready to be issued. This is a common occurrence in today's von Neumann type architectures since processors are much faster than memories and generate requests faster faster than they can be satisfied by the memory.

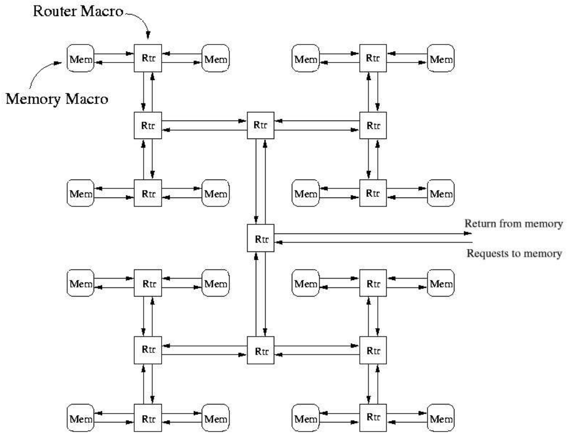


Figure 3.1. Basic H-memory Layout showing the organization of memory macros labeled “mem” and router macros labeled “rtr”.

3.1 Access Method

Access to the H-memory is parcelized, meaning each access is a self-contained series of bits with all data and control necessary for the request to be filled. Each access includes the address, an opcode signaling whether this access is a read or write, and, if appropriate, the data to be written. The companion to this address parcel is the select parcel which travels on a parallel wire in lockstep with the address parcel. This parcel is the same length as the write parcel and signals the presence of meaningful bits on the address line (figure 3.2). This parcelized access scheme is vital to the architecture because it allows accesses to take advantage of fine-grained pipelining throughout the memory since all data and state required for the access is, in a sense, local to the access and contained within the parcel [19].

The H-memory is organized as a binary tree with router macros at the inner nodes and memory macros at the leaves. Based on the first bit of the address encountered with the start of the select parcel, the select parcel is selectively routed to one of the current router macro's children. The address parcel is transmitted in both directions to all routers and all leaves below it in the tree. This is appropriate since only the presence of the select parcel activates the router and memory macros.

After the select parcel passes through a router, its leading bit is stripped off. The first bit of the new select parcel will correspond to the next bit in the address which will then act as the control for the next router encountered (figure 3.2).

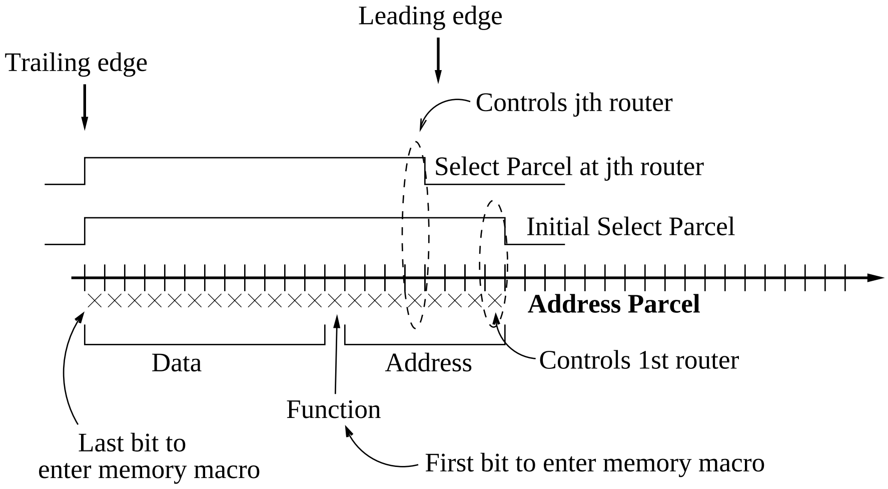


Figure 3.2. Change in parcels for an H-memory (12 bits/word, 256 word) as the parcels travel toward memory macro. At each router, the leading bit of the select parcel is stripped away.

Accesses enter and exit the memory from the root. The size of the access parcels are well defined. The address parcel consists of the address, the function (read/write enable), and a data word on a write. The layout of the memory insures that the time to access any word in memory is constant. These characteristics make the fine grained pipelining of this memory very natural and possible throughout the entire memory. It also makes interfacing to a finely pipelined microprocessor such as the Simple 12 explored by Niemier [4] easier since the access time is very predictable.

3.2 Router Macro

Router macros are located at the internal nodes of the H-tree and direct the incoming accesses down the path either to its right or left child based on the address bit that enters with the leading edge of the select parcel. The router macro is a serial device that continues the fine-grained pipelining through the memory. The full router macro is built from two "switch" circuits as discussed below and acts much as a railroad switch that sends a train down one of two diverging tracks. In addition, the router macro includes logic to strip off the leading bit of the select parcel in keeping with the access method discussed above (figure 3.3).

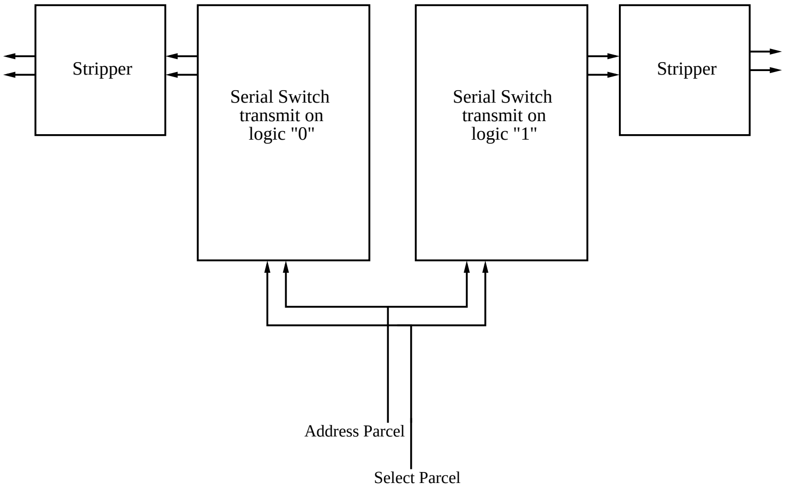

Stripper
Serial Switch transmit on logic "0"
Serial Switch transmit on logic "1"
Stripper
Address Parcel
Select Parcel

Figure 3.3. Components of a router macro. The serial switch modules selectively transmit the select parcel. The "stripper" module strips the leading bit from the select parcel after transmission.

3.2.1 The “Switch”

The device characteristics of QCA leads to different basic circuit design techniques and approaches. The “switch” discussed here is a direct outgrowth of these different characteristics and forms the basis for the H-memory.

The switch takes two wires as input, an address line and a select line. These names foreshadow their role in the memory structure that will be discussed. The values on the address line are transmitted regardless of the activity of the switch. The values on the select line are being “switched.” If the switch is “off”, the select line output is logic zero. If the switch is “on”, the output will be logic one as long as the select line input is one.

The select line can be thought of as an activation signal. The presence of a logic one on the select line signals that a switch decision is to be made. The decision is made based on the bit on the address line corresponding to the leading one on the select line. If this address bit is a logic “0”, the switch is left off. If the address bit is a logic “1”, the switch transmits the select bits.

On the select line, any number of logic ones can directly follow the leading one. The switch decision is made solely by the address bit corresponding to the leading one of the select parcel. The end of the select parcel is signaled by a logic zero on the select line. This resets the switch for a new switching event.

3.2.2 Full Router

The router macro is constructed from two serial switches plus circuitry to strip off the leading bit from the select parcel after it has been transmitted (figure 3.3).

The router macro takes advantage of the serial access parcels, and acts similar to a railroad switch that directs a train down one of two diverging paths. Based on the bit of the address parcel that arrives with the first bit of the select parcel, the access is shunted down one of the two paths. Since the address line is ignored in the absence of the select parcel, it is the select parcel that must be selectively routed to the right or left to route the access. By stripping off the leading bit of the select packet, the address bit is effectively stripped off as well. The next router encountered, then, will route the packet based on the next bit in the address. When the access reaches the memory macro, the select parcel accompanies only the opcode (read/write) and data to be written enter the memory macro (figure 3.2).

At the level of the tree closest to the leaves, a router macro connects two memory macros together (figure 3.4). The size of the memory can be built up by connecting two such half-H structures with another router macro (figure 3.5).

The necessary area penalty for the additional overhead is only the area of a single router macro. The additional time overhead depends on the size of the two subtrees being connected by the new router macro. This is due to the connection between layout and timing. As the size of the two subtrees to be connected grows, the distance to be traveled by the wires connecting them increases. The final delay will depend on implementation details.

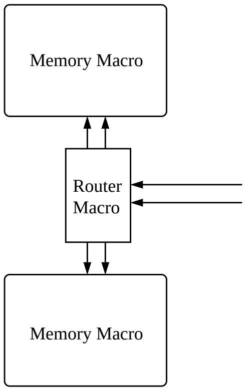


Figure 3.4. A single H at the lowest level of the tree shows a router macro connecting two memory macros. This figure shows the down-tree wires.

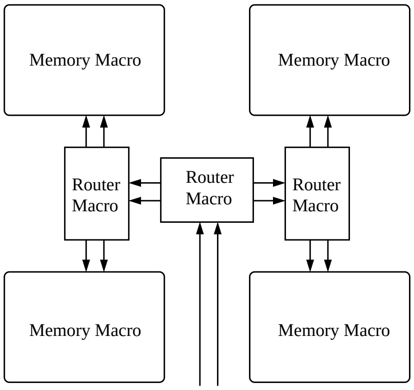


Figure 3.5. New router macro connecting two smaller Hs.

3.3 Return Path

The return routers in this basic memory are simply OR gates that join the two wires entering it into a single output (figure 3.6). Since the accesses are pipelined, all accesses require the same amount of time, and the output of all non-active memory macros is a logical zero, there will never be a collision between accesses that return data. In addition, the bitwise OR operation is trivially easy to implement serially requiring a single majority gate with one input locked to logic zero, allowing the fine-grained pipelining to continue on the outbound route of the memory.

3.4 Memory Macro

Each memory macro is a complete, serially-accessed memory unto itself, storing a single word of data. The memory macro consists of read and write enable logic that determines the function to be performed by an access, control logic to implement that function, and a data loop in which data is stored serially (figure 3.7).

The memory macro is accessed by a pair of serial parcels which travel on parallel wires. The first parcel contains the one bit operation code signaling whether the

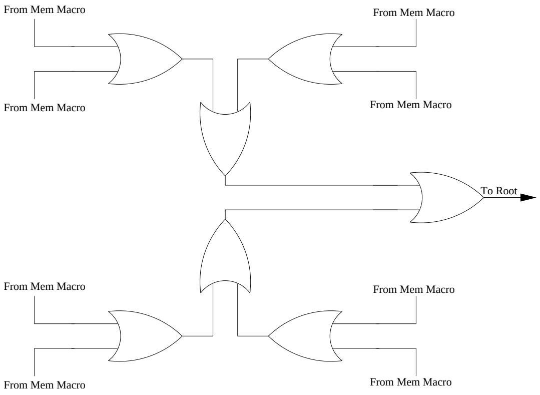


Figure 3.6. The return routers are OR gates whose inputs combine to return the data to the root of the memory.

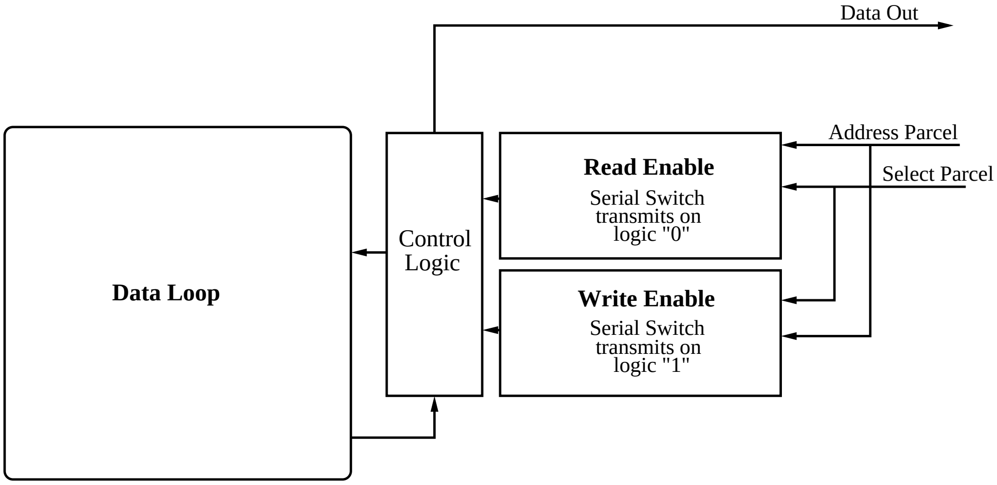


Figure 3.7. Components of a memory macro: two serial switches, the data loop, and minimal additional control logic that executes the read and write operations.

access is a read or a write followed by the word to be written if appropriate. The second parcel is the “select” parcel. These are, of course, the same parcels discussed with the router macros. By the time the address parcel reaches the memory macro, the leading edge of the accompanying select parcel corresponds to the start of the one bit opcode that signals whether the access is a read or write.

3.4.1 Read and Write Enable

The read enable and write enable logic is based on two of the serial switches discussed with the router macro. Because the memory macro has two functions, only one of which will be active at a time, the switches are a natural and efficient way to continue the fine grained pipelining into the memory macro itself. The one bit opcode that arrives with the leading edge of the select parcel, allowing the read and write enable logic to operate the same as the router macros.

3.4.2 Control

The control logic includes the data loop control and the data out control. The data loop control is responsible for either recycling the stored word back into the memory or replacing the stored bits with a new word to be written. The data out control outputs logic zeros unless the access is performing a read in which case the stored word is read out a bit at a time and sent serially up the output path to the root of the memory. By explicitly outputting logic zero on non-read cycles, the memory is guaranteed not to have any collisions on the outbound path, allowing the simple return routers discussed above.

Since the data is stored serially in the data loop, a serial write is very natural. As the data cycles around the storage loop, each bit is replaced in turn by a bit in the new word. After the entire word has been written, the new word is then allowed to cycle around the data loop refreshing itself. Similarly, a serial read is also very natural, copying each bit onto the output line as it passes through the head of the data loop.

3.4.3 Data Loop

There are several possible memory cells in QCA. For instance, there are straightforward analogues of DRAM, SRAM, and ROM cells (figure 3.8).

The ROM cell is based on the majority gate. If the east input is the stored bit, and if the north and west inputs are different, then the value of the stored bit will be output to the south. If the north and west inputs are the same, then that value rather than the stored value will be output (figure 3.8c).

The DRAM cell is a transitory latch and is implemented naturally by a QCA wire spanning several clocking zones. Data is inserted in one end of the wire and exits at the other end of the wire after a time delay of one fourth of the number of the clock zones the wire passes through (figure 3.8a).

A single bit SRAM-type cell that would be traditionally embedded into a 2D array can be implemented by looping a wire with four clock zones over onto itself with a majority gate joining the start and finish of the wire. The output of the majority gate, and the start of the wire, is on the east. The south input to the majority gate is the output end of the wire. The north and west inputs are the row and column selects. If the row and column bits are different, the value that was stored in the wire will be written back into the wire again. If the row and column select signals are the same, that value will be written into storage. The data is read out via a wire fanout where a cell in the hold phase drives more than one of its nearest neighbors to send the value down two to three paths (figure 3.8b).

The memory macros used in the H-memory store data by combining the SRAM and DRAM cells by expanding the wire loop of the SRAM cell to store multiple bits. By making the wire longer and passing through more clocking zones, the capacity of the memory cell can be expanded to hold an entire word, or multiple words. The simple majority gate control logic is replaced by a 2:1 multiplexor that allows the loop to either refresh its old data or write new data to the loop as each bit passes through the multiplexor and into the memory loop. Storing multiple bits per loop substantially reduces the control overhead required per bit stored.

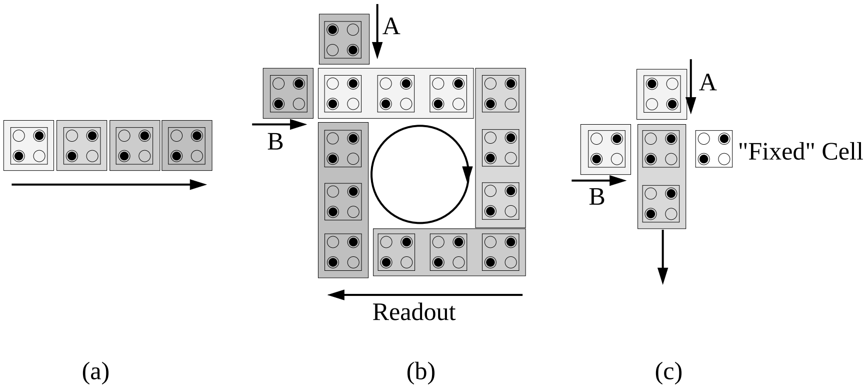


Figure 3.8. Three basic QCA memory cells. a) Translation of DRAM, b) Translation of SRAM. When A=B, a new value is written to the cell. A≠B, the previously stored value remains. c) ROM cell. If A=B, that value is output. If A≠B, the stored value is read out. [31]

The question then becomes how best to fold the wire, ultimately a consequence of the relationship between layout and timing. The answer depends on which of several possible factors is to be optimized. The most straightforward characteristic to optimize is area per bit stored. However, there are also other possible factors that may become important as the fabrication process becomes better defined and the constraints and challenges become clear. For instance, it may become important to minimize the number of clock zones, the number of 45 degree wires, or the number of wire crossings. Each of these would impact the final design.

An example of such a design is the spiral (figure 3.9). In this example, the wire is wound in a spiral. Since all data is moving in the same direction, clocking zones can be shared between layers of the spiral. To determine the number of bits stored in this spiral, one can count the number of clocking zones the wire passes through. In all, the spiral in 3.9 passes through 55 clock zones, and stores 14 bits. In this case, the number of clock zones is not evenly divisible by 4 because the first and last clock zones of the wire share a zone. An arbitrary number of bits can be stored in a single loop by adding another turn to the spiral. There is, however, an upper bound on the number of turns that can be made. This is due to the upper bound on the length of a QCA wire in a single clock zone. The outer turns have wider corners to turn than the inner turns, placing more cells in the "corner zones." These outer turns will eventually limit the size of a single simple spiral. What this limit actually is will depend on the device implementation. This is by no means a fundamental limit on the number of bits that can be stored in a single wire, though, only on a single continuous spiral. To store more bits in a spiral configuration, the wire could be folded into multiple, small, cascaded spirals before bringing the ends of the wire back together (figure 3.10).

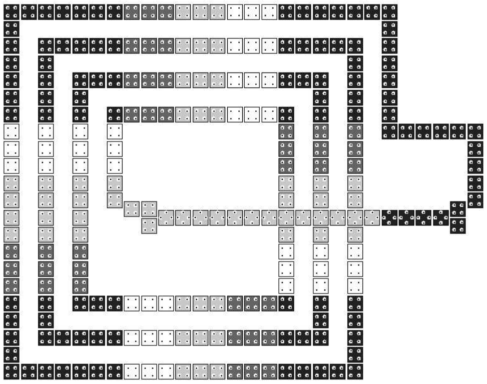


Figure 3.9. Spiral data loop storing 14 bits that circulate in a clockwise direction. Here, the ends of the loop are shown to join together directly without intermediate control logic. The turns of the spiral are made of 90 degree wires. The segment of wire closing the loop is a 45 degree wire, allowing the signals to cross over eachother without interference. Cell shading indicates the clocking zones.

3.5 Conclusion

This basic H-memory is a departure from traditional 2D array architectures. The important features of the architecture are serial access, uniform access time, oppor-

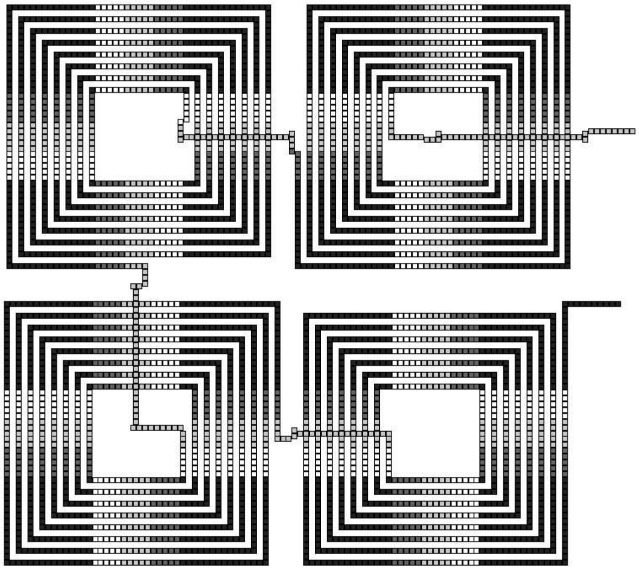


Figure 3.10. Cascaded data loops allowing arbitrary numbers of bits to be stored in a single wire.

tunity for fine-grained pipelining within the memory itself, scalability, and finally the potential high density of the memory.

Serial access is a key aspect of the architecture. Serial access avoids timing problems associated with parallel access, increases the percentage of area available for data storage, and utilizes the inherent latching of QCA wires. The timing problem stems from the parallel access requiring turning corners with several wires. The outside wires would travel significantly farther than the inside wires, leading to an explicit delay in the outer wires. Accessing the memory serially avoids this cornering problem by reducing the number of wires bundled together. In addition, serial access reduces the amount of space in the design required for wiring which increases the area available for data storage. Finally, serial access is simple in QCA wires since the wires act inherently as latches, requiring no additional circuitry to support the serial movement of data, and finally serial access complements the basic storage mechanism of the memory macro – the data loop. Rather than forcing an awkward parallel access on the structure, the serial access exploits the simplicity of the required serial logic.

Uniform access time is a necessary part of an efficient memory in QCA. There are two levels at which to consider this. First, once a request enters the memory, it should take the same amount of time to access any word in the memory in order to facilitate the finegrained pipelining of memory accesses and exploit the full potential of QCA wires. In other words, by guaranteeing uniform access time, the lack of collisions on the return path is guaranteed. The second level is that in interfacing with a QCA-based microprocessor such as the Simple 12 [4], it is essential to have predictable memory access times because of the very fine-level of pipelining within the processor. Finally, the opportunity for fine-grained pipelining within the memory itself is an advantage over traditional 2D array designs in which accesses are pipelined up to the memory but not into the memory itself. The full impact of this potential will be best felt when memory requests can be overlapped in the memory. This potential is available, though, because the H-architecture allows the benefits of pipelining to be fully exploited by carrying the pipelining into and throughout the memory.

CHAPTER 4

H-Memory Implementation

The overall H-memory architecture was discussed in the previous chapter. This chapter uses a particular circuit design to compare the potential performance of the H-memory to projected RAM performance. This circuit design is by no means optimized. It is a proof of concept design that illustrates the potential of the architecture and the utility of designing to take advantage of the new characteristics of QCA. To consider where future designs should focus, consider the components of the memory latency (figure 4.1). The access time will be discussed in more detail later in the chapter, but it can be seen where the focus of future designs should be. For very small memories, the router delay dominates the access time. As the memory capacity grows, though, the wire latency quickly becomes the dominant factor in the memory latency. For more space efficient layouts, the distance needed to be traversed by wires will decrease, improving the performance of the memory.

4.1 Assumptions

Circuit densities depend on the implementation details such as cell size, clocking wire pitch, and QCA wire pitch (minimum separation of QCA devices carrying unrelated signals). The final appropriate assumptions are yet to be determined and will depend on the precise implementation of the QCA cell and clock wiring. For instance, the cell to cell spacing, the number of cells per clock zone, the pitch of

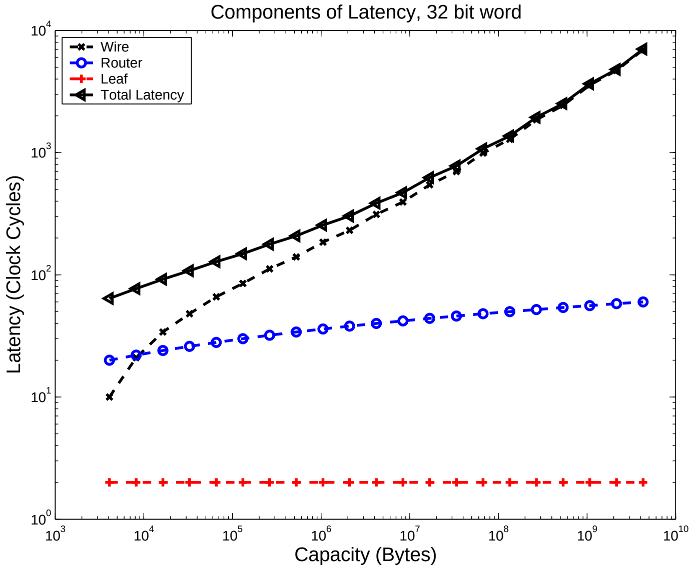


Figure 4.1. Components of latency as the memory capacity grows for a 32 bit word and a varying number of memory macros.

QCA wires, and the organization of clocking zones are all subject to fabrication details and decisions.

Several molecules are being explored for use as molecular QCA cells. Room temperature device operation is estimated for QCA molecules between 1-10 nm in size. The most promising molecules are two-dot cells with inter-cell distances ranging from 1 nm for the Creutz-Taube ion to 2.5 nm for a silicon phthalocynanine dimer [61]. Two of these two-dot molecules are required to build the four-dot QCA cells on which this work is based. In this four-dot cell context, then, the inter-cell distances range from 2.0 nm to 5.0 nm. A cell size of 2.0 nm will be assumed here.

Another parameter that the molecule choice effects is the longest wire that can successfully switch in a single clock zone. A conservative estimate of a maximum of 1000 cells per wire in a single clock zone were used. Estimates for the molecules discussed above suggest that at room temperature the maximum is approximately

$10^{13} - 10^{14}$ cells per cm^2 [34]. The maximum clock rate is determined by the time it takes for a single electron to move across a molecule. In the molecules being examined, this results in an upper limit on the order of $10^{16} Hz$ [5]. This implies operation in the terahertz may be possible. The final density and clock rate will be tempered somewhat by thermal considerations that require further study before they can be fully quantified.

The clocking wire implementation also effects the possible density that can be achieved. For these designs, the clocking wire size limitation was taken into account by the clocking zone size. The memory and router macro designs discussed here were designed assuming a minimum clocking zone of 8x8 cells. The 8x8 clocking zone size was chosen to somewhat reflect today's fabrication technology as well as the technology expected in the near future. Clocking wires with a width of 10 nm can be fabricated today. If a QCA molecule of width 2.0 nm is used, a width of 8 cells translates to 16 nm. Although this estimate ignores clocking wire pitch and connections to sources that drive the clocking signals, the 8x8 choice roughly corresponds to potential sizing constraints.

Finally, the pitch of QCA wires will also effect the density of circuits. For this work, a separation of 1 cell's width is assumed.

4.2 Data Loop

The delay in the data loop is vital to the operation of the memory since the delay directly corresponds to the number of bits stored in the loop. As a result, speed is not a factor to be optimized. Area remains as a key factor to optimize, and the potential ease of implementation is also a factor to be considered.

4.2.1 Parallel Wires Form an Upper Bound

A single bit is required to traverse four clocking zones in order to store a value. Obviously, this is the minimum required for a self-refreshing loop. It can also be thought of as the minimum for a transitory latch in a wire. Since the storage is accomplished by the value in a segment of wire in a “hold” state, any communication of that data or the transfer of that data to the next “latch” requires the complete four clock zones to be traversed.

An upper bound on the density of a storage loop can thus be estimated as the number of bits that could be stored by densely packed parallel wires in a given area (figure 4.2). This assumes all storage is done by transitory latches and with no regard to access or organization, but it is a useful measure of the maximum bits stored in a given area. This is because this setup maximizes the sharing of clocking zones between bits, and minimizes the number of cells per wire since each wire takes the most direct path to the next clock zone, which is a straight line. This calculation depends on the minimum clocking zone size and the minimum wire separation. More precisely,

$$\frac{bits}{area} = \frac{1}{4 * p_{QCA} * cz_w} \tag{4.1}$$

where “cz_w” is the minimum clocking zone width, and “p_{QCA}” is the minimum center to center separation between adjacent QCA wires.

This heuristic shows the potential storage density resulting from allowing multiple stored bits to share clocking zones. To compare this to a storage strategy that does not take advantage of shared clocking zones, consider the area required to store a bit without sharing,

$$\frac{bits}{area} = \frac{1}{4 * cz_w^2} \tag{4.2}$$

For instance, in figure 4.2, the minimum clocking zone size is 3 cells wide, and the minimum wire spacing is one cell between wires. These numbers are chosen solely for illustration purposes. Assuming a QCA cell with a 2.0 nm cell-to-cell distance, the clocking zone width is 6.0 nm, and the QCA pitch is 4.0 nm. The upper bound for this clock zone width, allowing clocking zone sharing, is 96 nm^2/bit as opposed to 144 nm^2/bit required for a single latch (no sharing). The improvement in area per bit is achieved by sharing clocking zones for multiple bits.

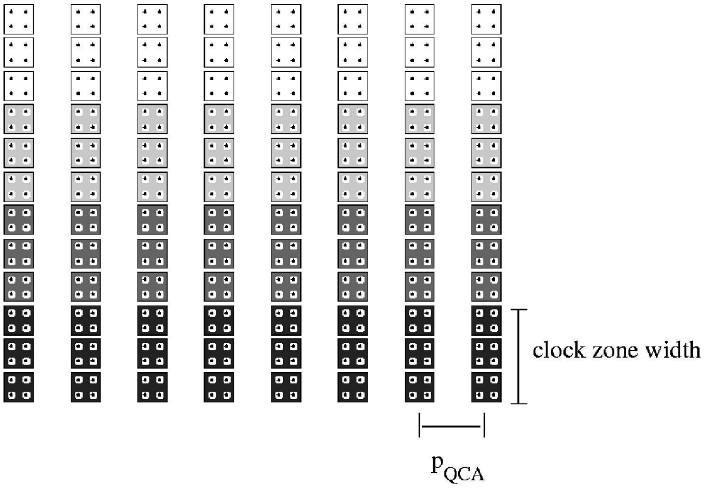


Figure 4.2. Given an area and a minimum clock zone size, an upper bound on the capacity of a storage loop can be identified.

4.2.2 Loop Configurations

There are several potential loop configurations. The most useful in this discussion are the flattened spiral (figure 4.3) and the standard spiral (figure 4.4). The flattened spiral has two sides that each have at least one bit stored per side per turn. The other two sides are caps that consist of a two clock zones in the same phase. The caps allow the inner turns of the spiral to be as close as the QCA pitch allows. However, the number of turns is limited by the length of the longest wire in the caps (i.e. the outside turn). The standard spiral has one bit stored per side per turn. The outside turn again limits the number of turns in the spiral. However, the distance traveled around a corner by the outside wire is only half that traveled through a cap in the flattened spiral. Because of the one bit per side property, the center of the standard spiral is left open, but the number of turns per spiral can be greater than that of the flattened spiral.

There are two sets of equations to consider in calculating the area required by the loop configurations. The first is the area required by the looped wires themselves. The second is the area required by the the wires and the clocking zones. The QCA wires only measurement speaks to the density of the loop itself. Including the clocking zones speaks to how closely different loops may be packed or how closely the storage loop can be placed to other circuitry without having the clocking zones interfere with eachother.

For the QCA wire only calculation, the height and width of the flattened spiral are calculated as follows:

$$height = p * (2n - 1) + x * (4z - 1) \tag{4.3}$$

$$width = 2p * (n - \frac{1}{4}) \tag{4.4}$$

With clocking zones, the equations change slightly:

$$height = 2c * \lceil \frac{p * (n - \frac{1}{2})}{c} \rceil + c * (4z - 1) \tag{4.5}$$

$$width = 2c * \lceil \frac{p * (n - \frac{1}{4})}{c} \rceil \tag{4.6}$$

Finally, the number of bits stored in the flattened spiral is:

$$bits = 2 * z * n \tag{4.7}$$

For the regular spiral, the height and width are the same since it is a square. The height for the QCA wire only calculation is calculated by:

$$height = p * (2n - 1) + c * (4z - 1) \tag{4.8}$$

Including the clocking zone area, the height equation is calculated by:

$$height = 2c * \lceil \frac{p * (n - \frac{1}{2})}{c} \rceil + c * (4z - 1) \tag{4.9}$$

The total number of bits stored is calculated by:

$$bits = 4 * z * n \tag{4.10}$$

In all of these equations, c is the clocking zone width, n is the number of loops in the spiral, p is the center-to-center distance between QCA wires, and z is the number of bits stored in one loop of the straight section of wire connecting the two caps. For instance, in the flattened spiral in figure 4.3 and the standard spiral in figure 4.4 z is equal to one.

In the figures shown, the minimum clocking zone size is five cells by five cells with a minimum wire separation distance of one cell. Using the simple parallel wire heuristic, a bit requires 160 nm^2 using these parameters, as opposed to 400 nm^2/bit without sharing zones. As the minimum clocking zone size grows, then, the benefit of sharing clocking zones increases (figure 4.5). The flattened spiral here stores one bit in 166 nm^2. The spiral stores one bit in 208.3 nm^2. While the flattened spiral has a higher density, the standard spiral has a few advantages. Most significantly, our collaborators working on the devices believed the standard spiral to be more easily implemented. The standard spiral is used for the area calculations that follow since it is believed to be more realistic. However, as the fabrication process matures, higher densities approaching the bound determined by this simple heuristic may be achieved.

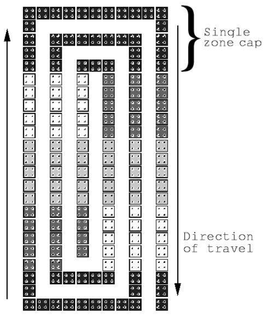


Figure 4.3. The flattened spiral configuration for a data loop drawn with a 5x5 cell minimum clocking zone, and a 1 cell minimum wire separation. There is at least one bit stored on each not cap side per turn. The cap zones consist of one zone, the size of which is constrained by the longest wire passing through it (i.e. the outside turn of the spiral).

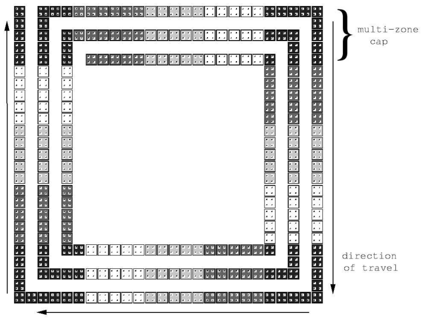


Figure 4.4. The standard spiral configuration for a data loop drawn with a 5x5 cell minimum clocking zone, and a 1 cell minimum wire separation. There is one bit stored on each side per turn.

4.2.3 Closing the Loop

To form a self-refreshing loop, the ends of the spiral must be connected. Because wires can cross on the plane, this is done by moving one end to a 45 degree wire

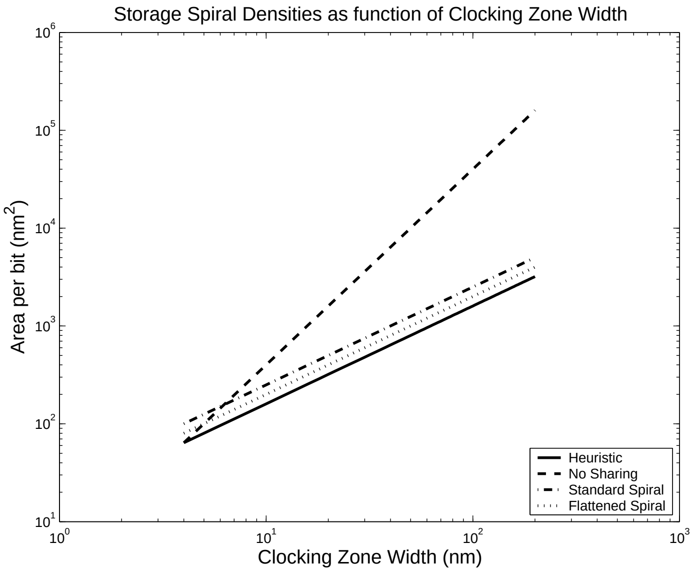


Figure 4.5. Area required per bit for the heuristic, no clocking zone sharing and two spiral configurations as a function of clocking zone width. As the size of the clocking zone grows, the benefit of sharing clocking zones for multiple bits increases.

that can cross over the 90 degree wires of the spiral turns. When the data loop is incorporated into the memory macro, the ends of the spiral are connected through a control logic circuit that allows the memory macro to either refresh the loop with the old data or replace it with a new word.

4.3 Macro Implementations

As discussed in the previous chapter, the H-memory is built of memory macros and router macros. The memory macro is a self-sufficient memory with read and write logic incorporated into it. Router macros with strippers on the select line outputs are used to build larger memories.

4.3.1 Memory Macro

The memory macro consists of the data loop, read enable, write enable, and control logic. The control logic includes a 2:1 multiplexor that controls whether data in the data loop is circulated back into the loop or is replaced by a new word. This decision is controlled by the write enable signal. The other piece of the control logic is simply an AND gate that controls whether the data circulating through the data loop is written to the output or if the default “0” is output. This decision is controlled by the read enable signal. The read and write enable are implemented by a router. The bit that enters the memory with the start of the select parcel is the operation code bit that determines whether the operation to be performed is a read or a write. A logical “1” signals a write. A logical “0” signals a read.

Remember that the select parcel that accompanies the data parcel is generated when the request enters the memory. When the request enters the memory macro, the select parcel is the size of one word. Trailing the select parcel is a logical “0” bit that signals the end of the request. This allows the memory macro to know when the end of the request has entered the macro and when it should stop operating on its word.

Figure 4.6 shows the layout and schematic of the first cut memory macro implementation. The memory macro in the figure stores 12 bits. The size of a single simple spiral is limited by the length of the wire in a single clocking zone. The maximum such wire is the outside turn of the spiral. To bring the designs somewhat in line with some of the predicted constraints that will be placed on fabricate-able systems, the number of bits stored in a single loop is limited to 32 bits. To store larger words, multiple spirals are cascaded (figure 4.8).

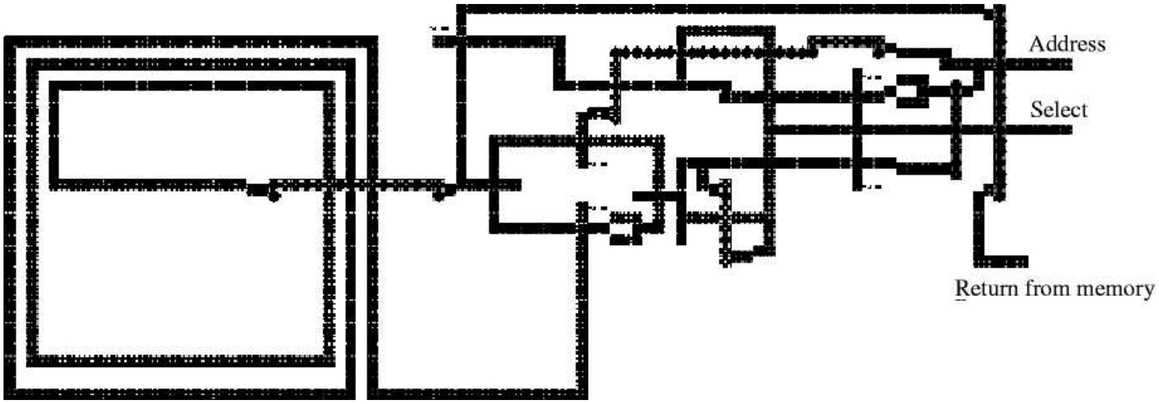


Figure 4.6. Memory macro layout. See figure 4.7 for schematic and boolean equations describing memory macro operation.

4.3.2 Router Macro

Router macros consist of two serial switches each with a stripper module on the output that replaces the leading “1” of the select parcel with a “0”. The implementation used for the comparisons can be seen in figures 4.9 and 4.10. Each bit requires two clock cycles to traverse the switch and a single clock cycle to traverse the stripper module. Because the stripper module can be implemented in wire along the path the parcel must already necessarily travel, its delay is counted as part of the wire delay (figures 4.11, 4.12).

Similarly to the function of the select parcel in the memory macro, when the trailing zero that follows the select parcel enters the router macro, the router macro is signaled to reset its state and be prepared to route a new request. The state in this case is transitory, consisting of a single latch.

4.4 Comparisons

There are three aspects of memory against which to compare the H-architecture: area, access time, and bandwidth. The H-memory is focused on the area aspect.

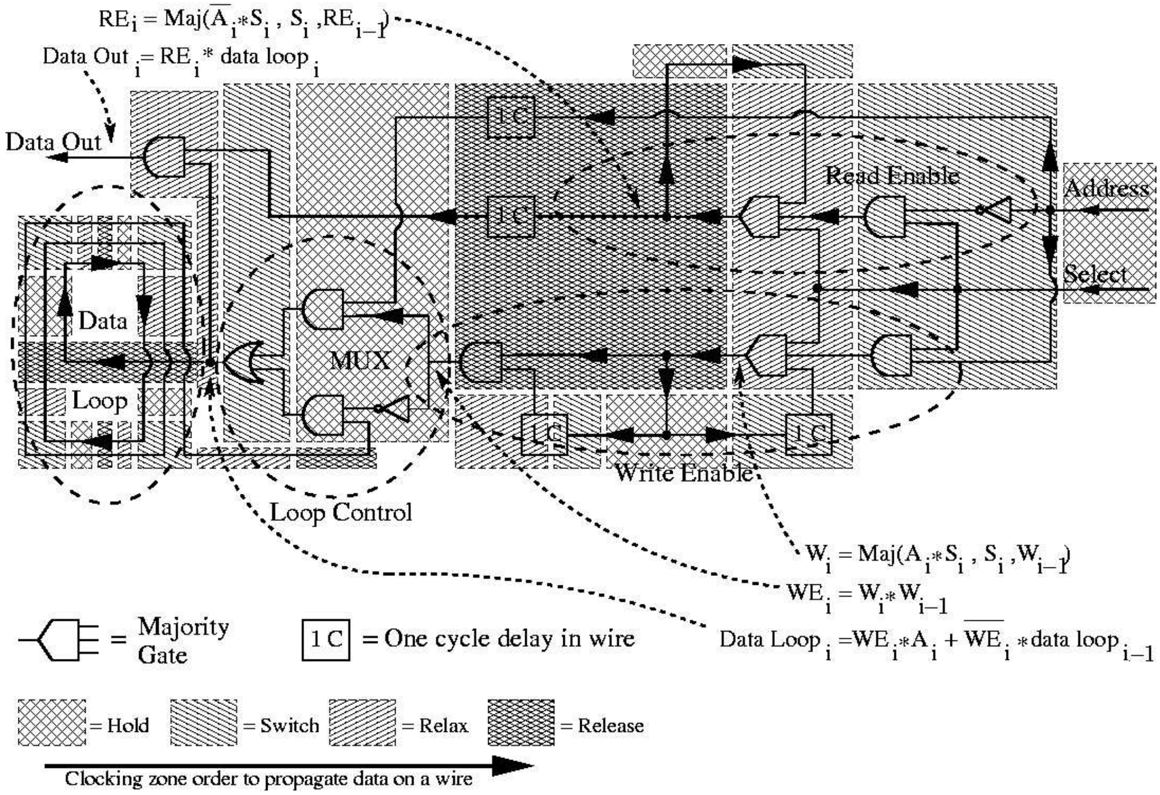

$RE_i = Maj(\overline{A}_i * S_i , S_i , RE_{i-1})$
$Data\ Out_i = RE_i * data\ loop_i$
Data Out
Data
Loop
MUX
1 C
Read Enable
Address
Select
Write Enable
Loop Control
$W_i = Maj(A_i * S_i , S_i , W_{i-1})$
$WE_i = W_i * W_{i-1}$
$Data\ Loop_i = WE_i * A_i + \overline{WE_i} * data\ loop_{i-1}$
= Majority Gate
1 C = One cycle delay in wire
= Hold
= Switch
= Relax
= Release
Clocking zone order to propagate data on a wire

Figure 4.7. Memory macro schematic. Shaded areas indicate clocking zone. Once a wire crosses a clocking zone boundary, its data is effectively latched.

However, it is important to evaluate it on the other aspects as well. As far as access time is concerned, the H-memory is relatively slow. The inherent latching in the wires that allows the fine-grained pipelining adds a high latency in number of clock cycles compared to CMOS wires, although in terms of nanoseconds, the latency may be lower. For bandwidth, due to that same fine grained pipelining, the bandwidth will be shown to be competitive with the next several DRAM generations.

In all of the discussions that follow, unless specifically noted otherwise, the parameters in table 4.1 are used.

Table 4.1

Parameters Used For H-memory Comparisons

Parameter	Value	Description
l_x	$58 + 54 * \lceil\sqrt{\frac{w_l}{32}}\rceil$	x dimension of memory macro, control logic + max bits per loop * number of loops cascaded in x direction
l_y	$33 + 54 * \lfloor\sqrt{\frac{w_l}{32}}\rfloor$	y dimension of memory macro, control logic + max bits per loop * number of loops cascaded in y direction
r_m	72 cells-width	maximum dimension of router macro
cz_w	5 QCA cells-width	minimum clocking zone width
p_{QCA}	$4.0nm$	Center-to-center distance between adjacent QCA wires
QCA cell size	$2.0nm$ x $2.0nm$	
max cells/wire	1000	Maximum number of QCA cells in a continuous wire
w_d	1-6942 cycles	Delay incurred traveling through down-tree wires during a memory request
w_l	64 bits	Data word length
n_r	$\log(n)$	total number of routers encountered by a parcel during a single memory request, n is the number of memory macros in the memory.
r_{lat}	2 cycles	Delay in cycles to travel through a router macro
$sync_lat$	w_l	Maximum delay required to insure request arrives with the start of the stored word
l_{lat}	2 cycles	Delay in cycles to travel through memory macro control logic
w_u	w_d	Delay incurred travel through up-tree wires and return routers during a memory request

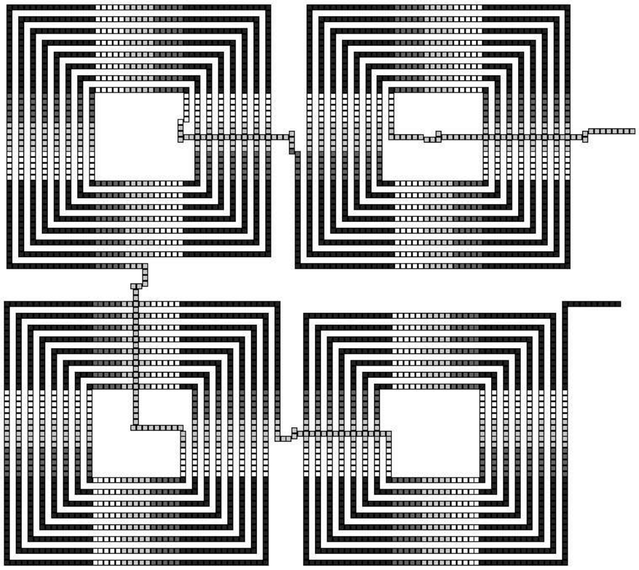


Figure 4.8. Data loop containing more than 32 bits.

4.4.1 Area Comparison

The area of the H-memory is calculated as follows:

$$area = [2^{\lfloor \frac{log_2(n)+1}{2} \rfloor} * (l_x + r_m) - r_m] * [2^{\lfloor \frac{log_2(n)}{2} \rfloor} * (l_y + r_m) - r_m] \qquad (4.11)$$

where "n" is the number of memory macros in the memory, "l_x" is the length of the x dimension of the memory macro, "l_y" is the length of the y dimension of the memory macro, and "r_m" is the length of the router macro's maximum dimension. For the designs in this work, $l_x = 14 * cz_w$, $l_y = 7 * cz_w$, and $r_m = 6 * cz_w$. The equation calculates the length of the x dimension of the memory and multiplies it by the length of the y dimension of the memory. The total area calculated, then, includes the wasted white space cost incurred due to the size and shape of the macros and the way they fit together in the layout (4.13). In the $\lfloor \frac{log_2(n+1)}{2} \rfloor$, the +1 is because the H structure doubles in the horizontal direction before doubling vertically. If the $\log_2$ of the number of memory macros is odd, the memory will have

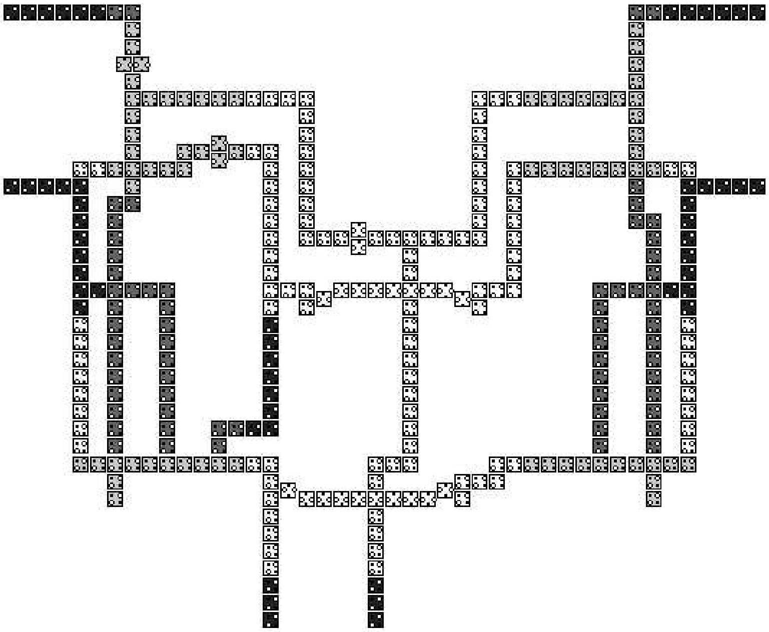


Figure 4.9. Router macro layout.See figure 4.10 for schematic.

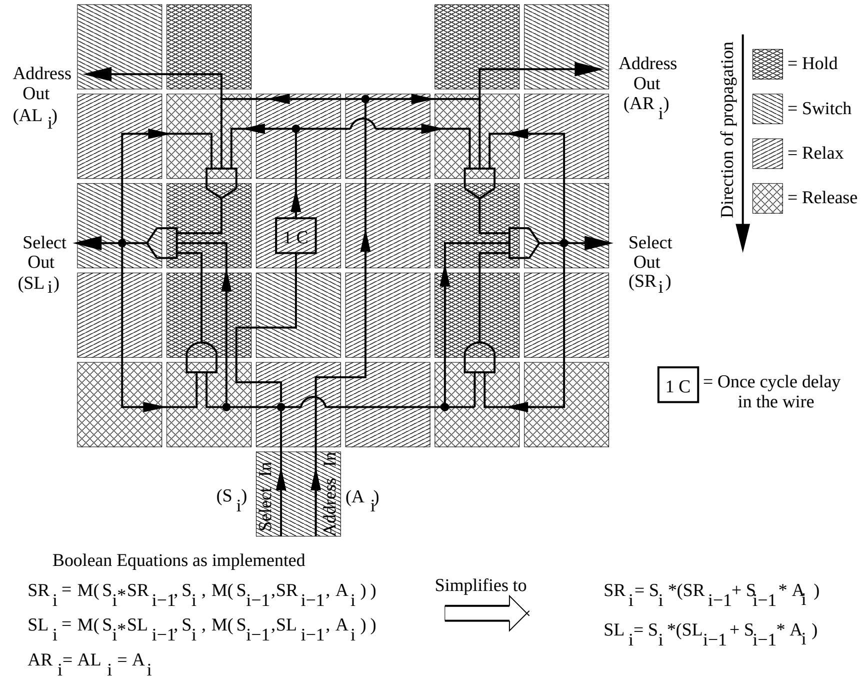
an oblong shape rather than a more square shape.

The densities of the molecular implementation of the H-memory has the potential to far exceed that of the end of the road map projections for DRAM and SRAM densities (figure 4.14). By 2018, the end of the road map, DRAM has a projected density of 58.21 $Gbits/cm^2$.

The density of the H-memory circuit implementations depend first on the implementation details discussed in the assumptions previously. In short, the density equals

$$density(\frac{bits}{area}) = \frac{n_m * w}{area} \tag{4.12}$$

where the numerator is the number of memory macros times the length of a word (i.e. the number of bits stored in the entire memory), and the denominator is calculated as described above.



Boolean Equations as implemented

$SR_i = M(S_i * SR_{i-1}, S_i, M(S_{i-1}, SR_{i-1}, A_i))$

$SL_i = M(S_i * SL_{i-1}, S_i, M(S_{i-1}, SL_{i-1}, A_i))$

$AR_i = AL_i = A_i$

$SR_i = S_i * (SR_{i-1} + S_{i-1} * A_i)$

$SL_i = S_i * (SL_{i-1} + S_{i-1} * A_i)$

Figure 4.10. Router macro schematic. The shaded areas represent clocking zones. Once a wire crosses a clocking zone boundary, its data is effectively latched.

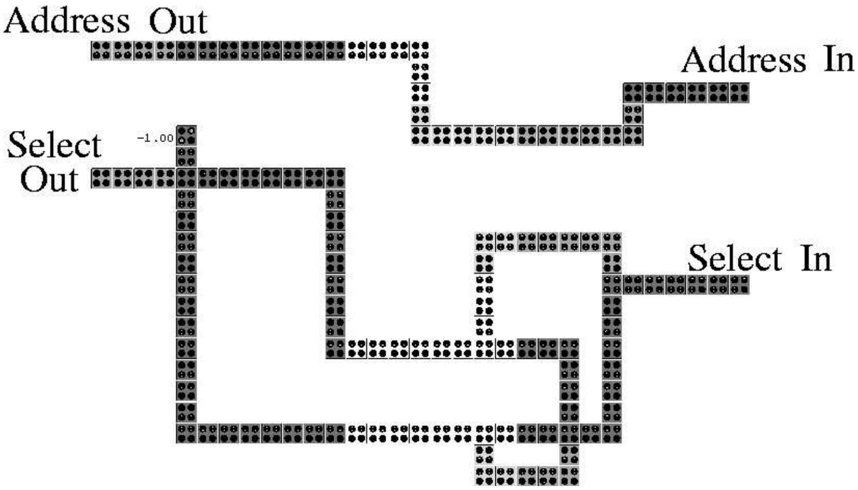


Figure 4.11. Stripper macro layout for removing the leading edge of the select parcel. See figure 4.12 for schematic.

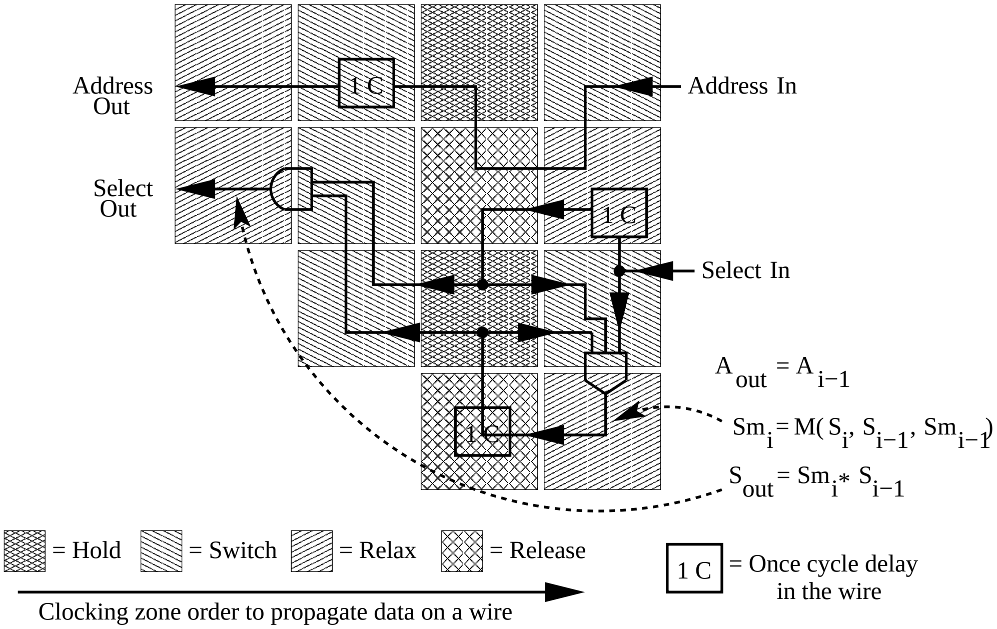

Address Out
1 C
Address In
Select Out
1 C
Select In
$A_{out} = A_{i-1}$
$Sm_i = M(S_i, S_{i-1}, Sm_{i-1})$
1 C
$S_{out} = Sm_i * S_{i-1}$
▩ = Hold
▧ = Switch
▨ = Relax
▩ = Release
1 C = Once cycle delay in the wire
Clocking zone order to propagate data on a wire

Figure 4.12. Stripper macro schematic. Shaded areas indicate clocking zone. If a wire crosses a clocking zone boundary, it's data is effectively latched.

Architecturally, the H-memory density depends on the ratio of storage to overhead, where the overhead includes the router macros and wires connecting the components. The density, then, translates to word size compared to the number of memory macros in the memory. For a 32 bit word, and 1 billion memory macros, the H-memory has a density of just 8.03 $Gbits/cm^2$. However, for larger words and the same number of leaves, the density improves, the density for a much larger word of 1024 bits, the density increases to 136.72 $Gbits/cm^2$. For the assumptions given at the start of the chapter, the maximum possible density, ignoring all overhead, is 363.2 $Gbits/cm^2$. This maximum is approached when the area used for data storage is large compared to the overhead of routers and control logic.

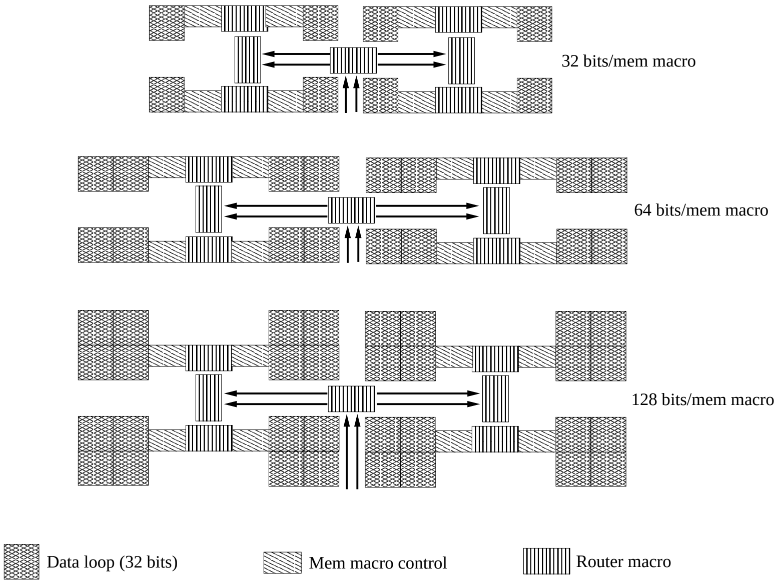


Figure 4.13. Memory macros and router macros shown in correct relative size for three data loop capacities. As the number of bits stored per memory macro grows, the size of the memory macro grows while the size of the router macro stays constant.

4.4.2 Effect of Brute Force Density Improvements on Latency

However, increasing the word length incurs a latency penalty as the worst case delay for the beginning of a word in the memory macro to cycle around to the head of the loop is the capacity of the loop.

$$sync_{lat} = w_l \tag{4.13}$$

In this equation, $sync_{lat}$ is the worst case latency in clock cycles that could be paid waiting for the beginning of the word stored in the memory macro to cycle around to the head of the loop. In the worst case, this is w_l, or the length of a data word. In addition, for relatively small memories, the larger word size can have a significant effect on the effectiveness of pipelining the accesses. For instance, if the path to the memory is much shorter than the length of the word in terms of clock cycles, there can be little effective pipelining. However, this is a minor consideration for reasonably sized memories.

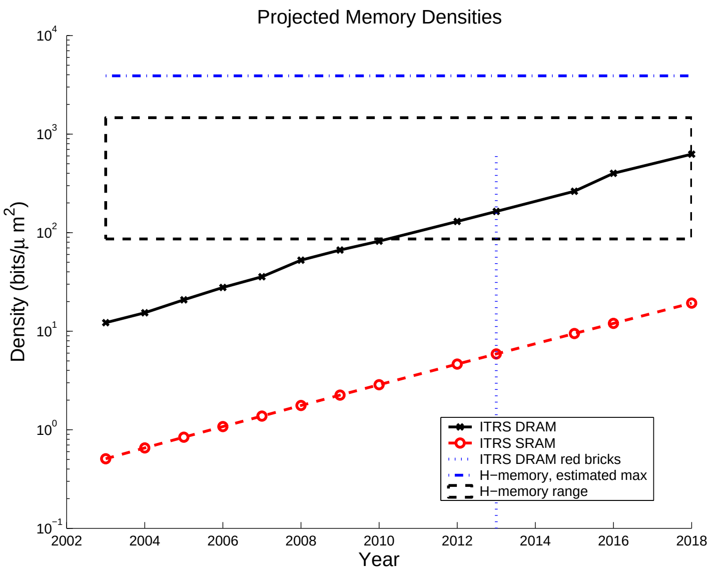


Figure 4.14. Density comparison between basic H-memory and ITRS projections for SRAM and DRAM. A range of H-memory densities for 1G of memory macros with between 32 and 1024 bits per word is shown. The maximum estimated H-memory density uses an 8x8 cell clock zone and the 2.0 nm cell. The ITRS "red brick wall" for DRAM is also shown.

4.4.3 Access Time Comparison

The other key parameter on which to judge the H-memory is the access time. The access time of the H-memory can be calculated as follows:

$$latency = w_d + n_r * r_{lat} + sync_{lat} + l_{lat} + w_u \tag{4.14}$$

"w_d" is the latency incurred by the traversing of the wires down the tree from the root to the memory macro. This is a function both of the size of the memory and of the number of cells allowed per clock zone. "n_r" is the total number of routers encountered by a single access. This is the log_2 of the number of memory macros in the memory. "r_{lat}" is the latency cost incurred by traversing one router macro.

The actual value of r_{lat} depends on the circuit design. In the design laid out in this chapter, r_{lat} is a delay of 2 cycles. "$sync_{lat}$" is the maximum penalty incurred for waiting for the start of the word stored in the memory macro to cycle around to the start of the stored word to cycle to the start of the data loop to synchronize with the start of the incoming access. This value is determined by the time required for the access to travel from the root to the memory macro modulo the size of the data stored at the memory macro. This synchronization delay requires no additional circuitry to handle it since the parameters involved, the size of the memory and the length of a word, are known and remain constant. All memory macros are synchronized together, and the time to travel to any memory macro is the same. "l_{lat} is the latency involved in accessing the memory macro itself, in other words, it is the latency required to traverse the control logic at the memory macro, and in this design is 2 cycles. Finally "w_u" is the latency traversing the wires back up the tree from the memory macro to the root. Because the return router is a simple OR gate, its latency can be included in the wire latency and incurs no additional timing penalties. Both wire latencies are determined by the maximum wire length allowed in a single clock zone and the distance that needs to be traveled.

There is no synchronization circuit at the leaves in the basic H-memory. Because all accesses require the same amount of time to travel from the root to the start of the memory macro, the synchronization can be done at issue time just above the root or by the processor. All that is required is a simple counter that is the word length and a buffer to store the thread until the counter reaches the appropriate value. Since all words are synchronized together (i.e. the start of each stored word reaches the start of the data loop at the same time), this simple scheme is all that is required.

It is interesting to note the contribution of each element to the overall latency.

The down-tree and up-tree wire delays are combined in this figure because reducing the latency of one direction reduces the latency of the other. For very small memories, the sync delay and the router delay are the largest components of the access time. As the capacity grows, though, the wire latency quickly begins to dominate the latency (figure 4.15).

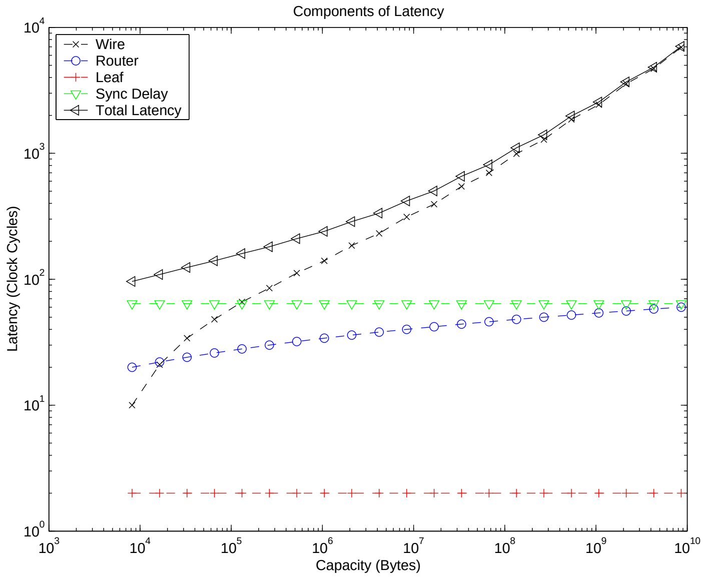


Figure 4.15. Components of latency as the memory capacity grows.

4.4.4 Bandwidth Comparison

The bandwidth (bits accessed per second) of the H-memory depends on the size of the word, the capacity of the memory, the clock rate, and the number of requests being made. The following calculation is for the maximum possible bandwidth, assuming there is always a new access waiting to enter the memory as soon as the previous thread passes completely through the root router macro. The capacity is important since the limiting factor in the bandwidth is how fast requests can be sent, and the address bits can dominate the size of the request parcel for large memories. Specifically, the bandwidth of the basic H-memory can be calculated as follows, where the denominator is just the size of the request parcel:

$$H\_bandwidth = (\frac{n * w_l}{p_{lat} + n * (parcel\_size + sync_{lat})}) * clock\_rate \tag{4.15}$$

The numerator is how much data is handled over n successive accesses. The denominator is the time required in cycles to access the data in the memory macros. This fraction multiplied by the clock rate gives the number of bits accessed per second. The size of the parcel is calculated by the word size plus the address size plus the one bit opcode and the required one bit space between successive accesses. The one bit space between successive accesses is the signal to the router macros that the access has passed all the way through the router (as discussed in section 4.3.1). The parcel size only includes the word on a store. However, assuming all accesses are stores gives the worst case of this best case bandwidth calculation. Because of the serial nature of the access and the in-wire latching, the number of bits in the parcel directly corresponds to the number of clock cycles an access occupies the entrance of the memory. The "p_{lat}" is the priming latency incurred by filling the "pipeline," or the time required for the first access to reach its memory macro and return. This delay need be counted only once since this delay is masked for all successive accesses. The *sync_delay* is the delay required to insure the access arrives at the memory macro at the same time the beginning of the stored word is at the control logic of the memory macro. This is calculated by the down-tree access latency modulo the size of the stored word. This equation is a best-case bandwidth description that assumes n accesses can be released into the memory successively with no delays between them.

The tradeoff concerning the bandwidth is word size and capacity. The larger the word and the smaller the capacity, the greater the percentage of the request parcel is actual data, and the larger the resulting bandwidth is. An additional consideration is the synchronization delay required to guarantee that accesses arrive in sync with the data circulating in the memory macros. This synchronization delay is the source of the sawtooth pattern seen in figure 4.16. When the memory is designed such that the delay for an access to travel from the root to the memory macro is a multiple of the size of the stored word, no additional synchronization is necessary, resulting in a higher bandwidth.

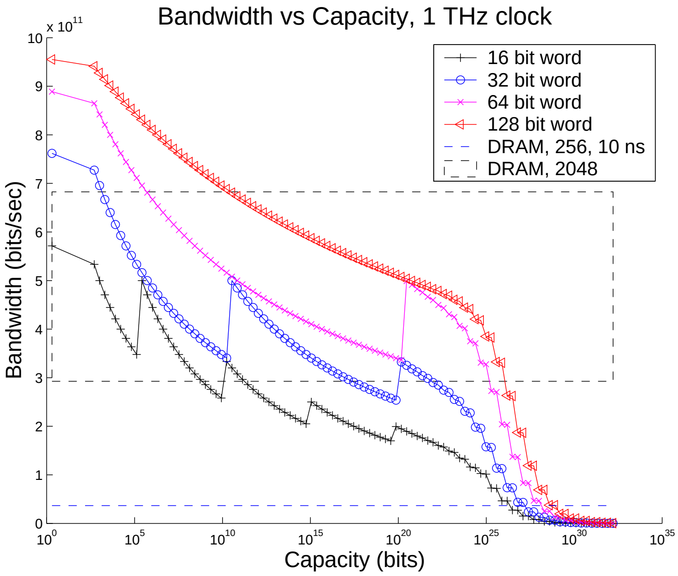


Figure 4.16. Bandwidth of the basic H-memory configuration for varying capacities and two estimates for DRAM bandwidth. The DRAM estimates assume 1/3 open row hit probability, a 1 ns open row access time, and a 10 ns random access time.

The DRAM bandwidth can be calculated in a similar fashion. Considering the raw bandwidth available within each memory macro, the bandwidth can be calculated as follows:

$$DRAM\_bandwidth = \frac{bits\_per\_access}{(t_{ra}) * (1 - p) + (t_{or}) * p} \tag{4.16}$$

where "t_{ra}" is the random access time, "t_{or}" is the open-row hit time, and "p" is the probability of an access hitting an open row. Once a row of the memory has been accessed once, it's data is at the sense amplifiers ready to be read again. This just-accessed row is considered an open-row. If the next access uses data from this same row, the access time is much faster.

If one assumes 2048 bits are available in the memory macro, a 10 ns random access time, 1 ns open-row hit time, and a probability of 1/3 of hitting an open row, the DRAM bandwidth is approximately 292 Gbits/sec. The actual number of bits available after each access is much lower at 256 bits. However, presumably researchers will find ways to make use of more and more of the bandwidth available in the memory macros. Future generations of DRAM will also find the access time decreasing, increasing the bandwidth. Figure 4.16 shows a range of bandwidths for DRAM, spanning from 10 ns access time to 2 ns random access time, assuming all 2048 bits are available in each access, independent of the capacity. Very high capacity, fast access time memories will be beyond the end of the road map, while lower capacity, slower access time memories potentially lie within the roadmap. The bandwidth for a contemporary memory making 256 bits available per access, and requiring 10 ns per random access is shown for comparison.

4.5 Conclusion

The implementation discussed in this chapter is a first cut implementation. Final device details will affect the circuit layout and precise area numbers. However, this implementation does demonstrate the great potential of the QCA H-memory. In terms of area, the H-structure exceeds end of the roadmap projections for DRAM density by a minimum of an order of magnitude. In terms of bandwidth, the H-memory far outperforms current DRAM bandwidth. Even assuming the most generous future for DRAM, the H-memory still has the potential for superior bandwidth. Only in terms of clock cycles per access does the H-memory lose to DRAM. However, after taking into account projected clock rates in the tens of terahertz, the H-memory is once again wins out over projected DRAM access times. There are, though, drawbacks to this straightforward implementation. Chapter 5 discusses a few enhancements to address these drawbacks.

CHAPTER 5

Enhancements

The main drawback of the H-memory is the long latencies incurred for larger memories. Due to the close connection between layout and timing, one solution to this is to move the stored data physically closer to the logic that will act upon it. There are two approaches to accomplish this. The first is to redesign the memory macro to store multiple words. This will make the tree shallower, reducing the distance traveled by an access to its data. The second approach is to incorporate the logic into the memory itself. This extreme processing in memory approach reduces the travel time of the accesses from memory to processor by placing the processing logic in the memory itself.

5.1 Multi-word leaves

The main source of latency in the "conventional" H-memory is the delay through the wires leading from root to leaf. As the memory grows larger, this problem is exacerbated (figure 5.1). The latency contributed by the wires can be calculated as follows:

$$wire_{lat}(clockcycles) = \frac{w_l}{max_{cz}}/4 \tag{5.1}$$

where $wire_{lat}$ is the latency in clock cycles, w_l is the total length of wire traversed, and max_{cz} is the maximum number of cells per clocking zone. The fraction is divided by 4 to give the result in terms of clock cycles. The contribution of the routers is a function of the number of routers an access passes through and the delay per router:

$$r_{tot} = log(n_l) * r_{lat} \tag{5.2}$$

where r_{tot} is the total latency incurred by routers, n_l is the number of memory macros, and r_{lat} is the latency in each router that is determined by the design. In the design used in this work, $r_{lat} = 2$. The final contributor to the latency is the control logic at the memory macro. This is constant regardless of the size of the memory since each access encounters only one memory macro. This latency is set by the design, and in this discussion, the memory macro latency is 2 clock cycles.

The main use of area is addressing logic (routers) and control logic within the memory macros. One way to minimize the delay through the wire is to shorten the wire. This can be done only by lessening the overhead circuitry it needs to weave around. The obvious solution is to store more than one word at each memory macro (figure 5.1). For instance, storing two words per memory macro reduces the area of the memory by half. Roughly, the area of the memory grows linearly with the number of memory macros. Remember from chapter 4 that the area is calculated by

$$area = [2^{\lfloor log_2(\frac{n+1}{2})\rfloor} * (l_x + r_m) - r_m] * [2^{\lfloor log_2(\frac{n}{2})\rfloor} * (l_y + r_m) - r_m] \tag{5.3}$$

where “n” is the number of memory macros in the memory, “l_x” is the length of the x dimension of the memory macro, “l_y” is the length of the y dimension of the memory macro, and “r_m” is the length of the router macro’s maximum dimension. For the first-cut design used for these comparisons, $l_x = x_{con} + x_{loop}$. The x-dimension of the memory macro control logic is 58 cells-width, and the basic storage spiral that holds 32 bits has an x-dimension of 54 cells. $l_y = y_{con} + y_{loop}$. In this case, y_{con} is 33 cells, and y_{loop} is 54 cells. If more than 32 bits is stored per memory macro, multiple copies of the basic storage loop are cascaded together, as discussed in chapter 4 (figure 4.8). These parameters are summarized in table 5.1.

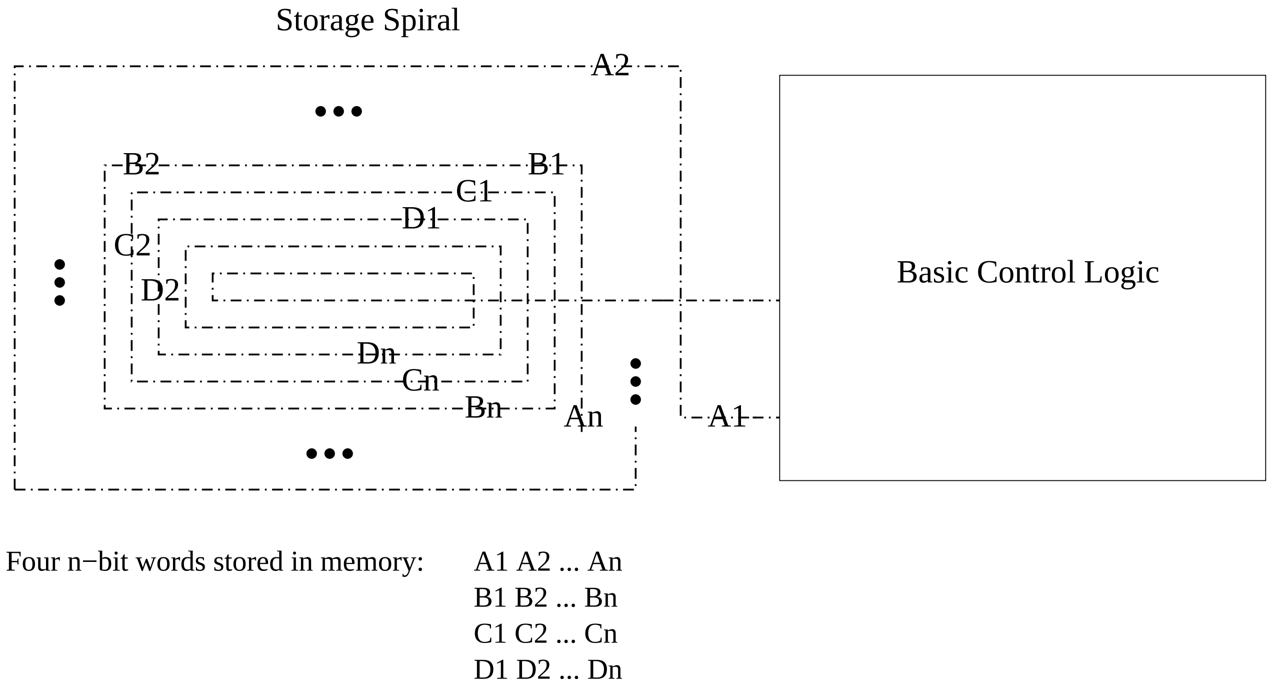


Figure 5.1. Alternative memory macro configuration to store multiple words in a single memory macro spiral. Maximum synchronization delay is the total number of bits in the spiral ($n_w * w_l$).

Provided the size of the memory macro does not grow faster than the number of bits stored per macro, reducing the number of memory macros reduces the distance to be traveled because it reduces the number of levels in the memory tree. Finally, due to the close connection between layout and timing, reducing the distance that needs to be covered by an access reduces the latency incurred due to wires.

However, this simple method significantly increases the worst case access time by increasing the worst case synchronization delay to access the correct word in the memory leaf to the total number of bits stored in the memory macro (figure 5.3).

$$sync_{wc} = w_l * n_w \tag{5.4}$$

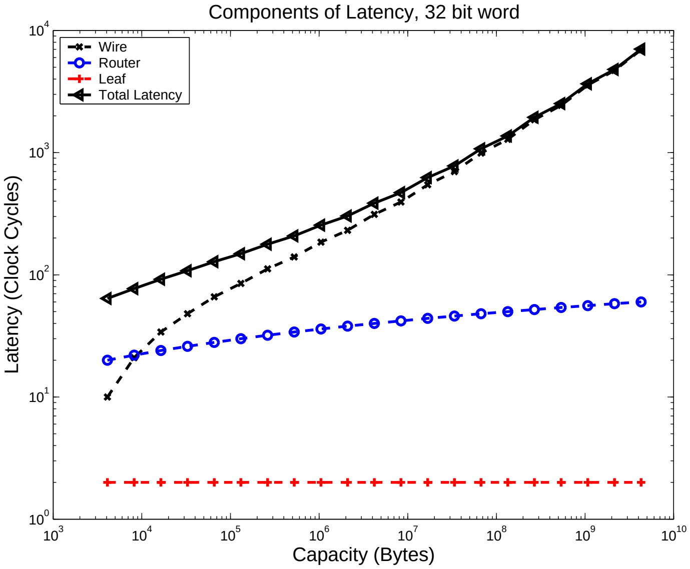


Figure 5.2. Components of latency as the memory capacity grows, 32 bits per memory macro.

where w_l is the length of a word, and n_w is the number of words stored per memory macro.

A more time efficient method is to borrow from the design of bubble memories. By storing each position in the word in a separate data loop, and storing several words at each leaf, the synchronization delay is limited to the number of words stored in each leaf rather than the total number of bits stored (figure 5.4). The tradeoff is that it requires more space than the straightforward method. However, the area required is still significantly less for large memories than the basic H-memory that stores only one word per leaf (figure 5.5) by trading increased control logic at each leaf for reduced numbers of routers.

5.2 Bouncing Threads Extension

Another approach to reducing the delay between a memory request leaving the logic and the requested data being returned is to incorporate the logic into the

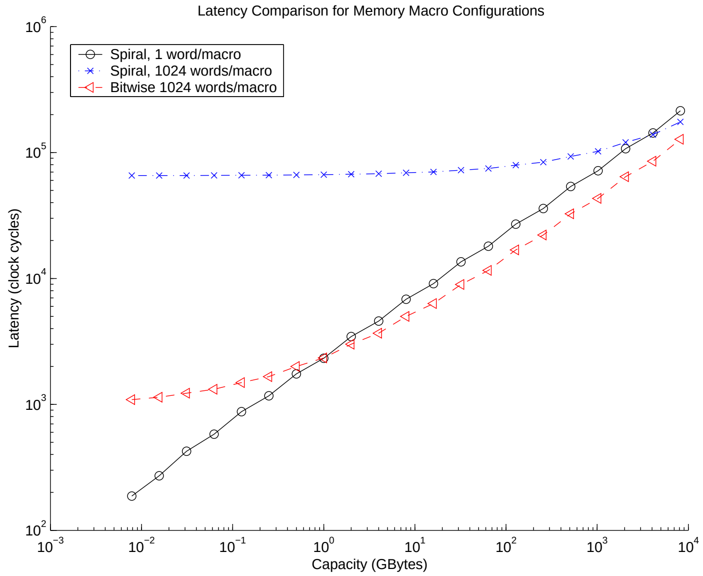


Figure 5.3. Worst case latency incurred for three memory macro configurations.

memory structure itself. One can envision incorporating simple ALUs into the memory structure to support simple operations to memory that take advantage of the memory access delay. This notion can be extended to moving the entire CPU functionality into the memory structure itself. This is an extreme extension of the processing-in-memory paradigm [12]. There are two options for incorporating the CPU functionality into the H-memory. The first is to place a simple but complete serial processor at each memory macro. The second is to distribute the pipeline stages (e.g. Fetch, Decode, Execute, WriteBack) throughout the router macros. These will be discussed in more detail below.

The extension from the basic H-memory is clear (figure 5.6). Each memory access parcel consists of an address, an opcode, and a word of data if the access is a write. As discussed earlier in this work, each memory access is self-sufficient, carrying with it all its necessary state. To support ALU operations in memory,

Table 5.1

Dimensions Used for Memory Configurations

Configuration	x dimension	y dimension
Spiral	$58 + 54 * \lceil\sqrt{\frac{w_l}{32}}\rceil$	$33 + 54 * \lfloor\sqrt{\frac{w_l}{32}}\rfloor$
Multi-word spiral	$58 + 54 * \lceil\sqrt{\frac{w_l * n_w}{32}}\rceil$	$33 + 54 * \lfloor\sqrt{\frac{w_l * n_w}{32}}\rfloor$
Bit-wise	$122 + 54 * \frac{n_w}{32}$	$64 * w_l$

w_l is the word length, n_w is the number of words stored per memory macro.

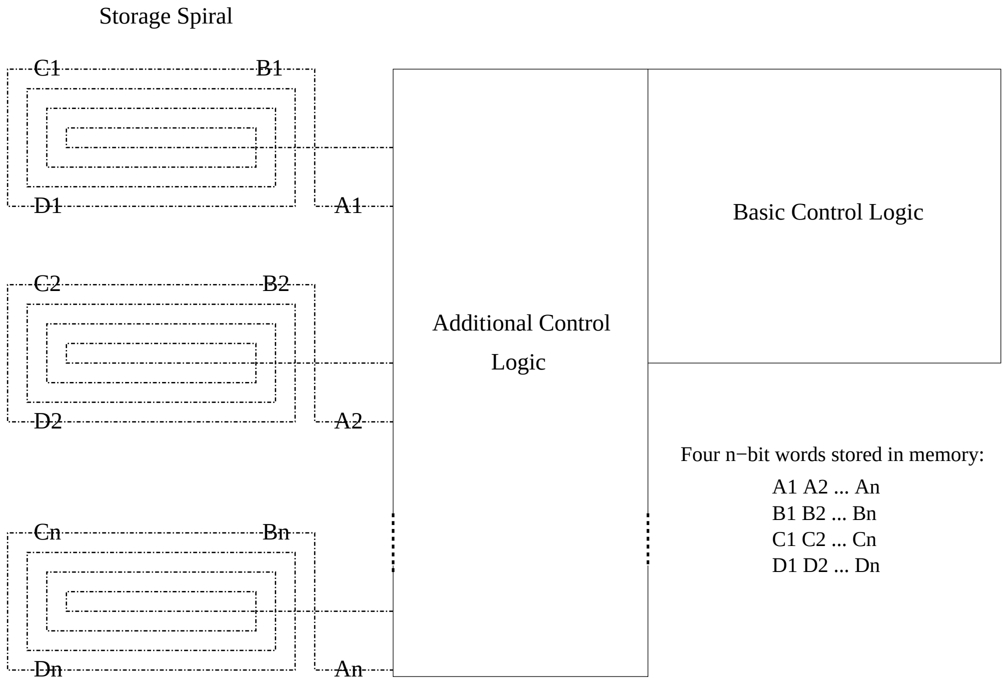


Figure 5.4. Alternative memory macro configuration based on bubble memory design to efficiently access one of multiple words stored in a single memory macro. Maximum synchronization delay is the number of words stored in the memory macro.

the data field for the write accesses could be replaced by an accumulator. One can imagine a mobile thread adding a program counter field and one or more registers to carry the necessary state with it.

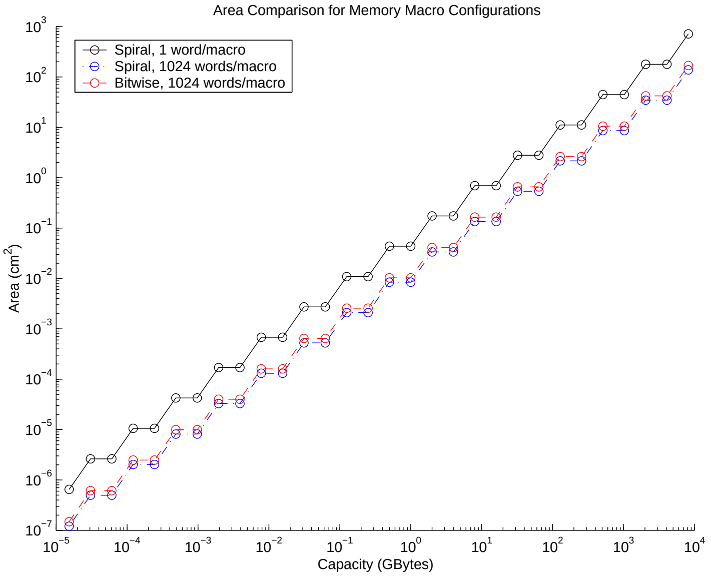


Figure 5.5. Area requirements for three memory macro configurations.

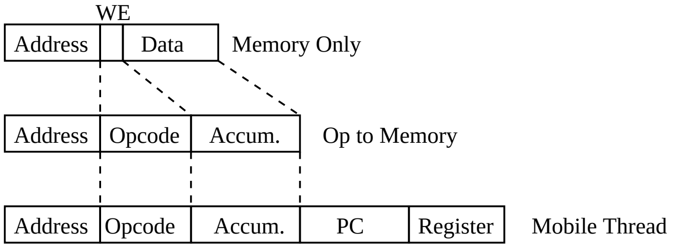


Figure 5.6. Fields for memory access, operations to memory, and independent thread [53].

5.2.1 Execution Model

The execution model has been explored by Giefer, et. al. [18] [22] [21] as well as initial ISA exploration [20]. In the bouncing threads execution model, instructions and data are stored at in the memory macros and processing logic is incorporated in the router macros. The research explored how to incorporate the logic in the router macros. The pipeline stages chosen to accomplish the processing is a matter for future research. For the preliminary exploration, the pipeline stages in the routers included: decode, execute, and pcinc (increment the program counter). The instruction fetch stage is implemented in the memory macros. These pipeline stages can be scattered and replicated throughout the memory (figure 5.7). The placement of each type of router in the H-tree has a significant impact on the execution time of the processes.

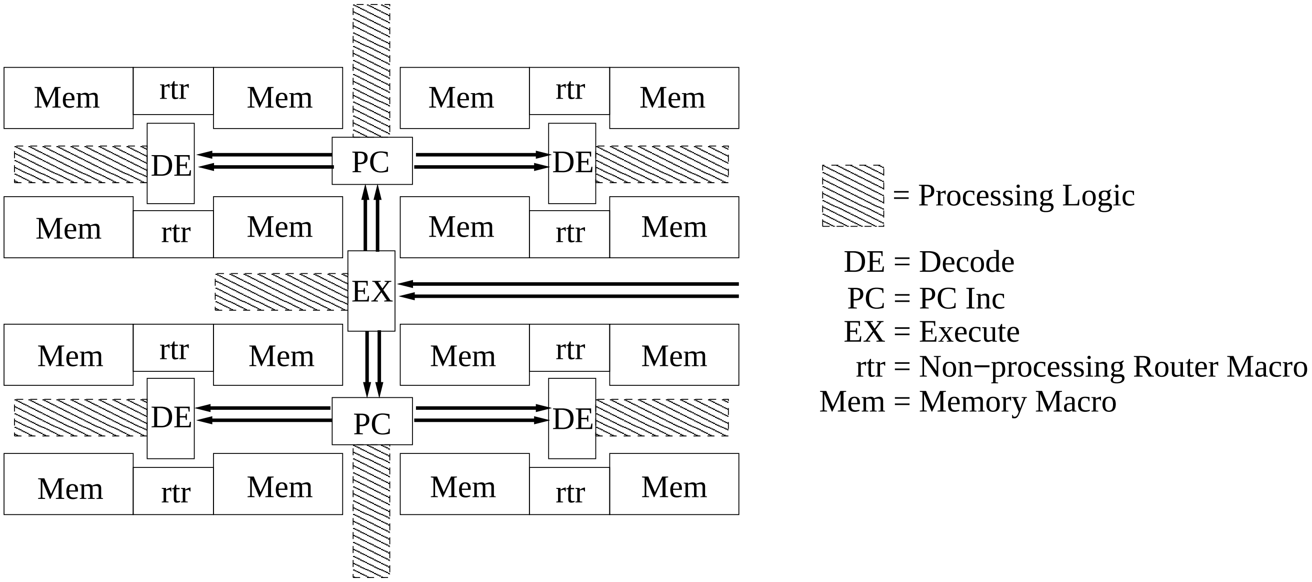


Figure 5.7. Sample processing router pipeline placement for bouncing threads execution model. Pipeline stages include iFetch (occurs at memory macros), program counter increment, decode, and execute. In this example, the lowest level routers are non-processing router macros. [21]

To execute programs, threads travel to a memory macro to pick up an instruction and then proceed to routers with the appropriate pipeline stage to execute, traveling to pick up data when necessary and then back to the necessary processing routers.

An alternative implementation of the bouncing threads execution model is to place complete processing logic at each memory macro. This is a potentially less efficient version that does not take full advantage of the processing-in-wire [46] potential of the structure, but it allows the exploration of other aspects of the memory/processor structure such as collision frequency and handling without the complications of the pipeline stage placement question. By placing the data and instructions far enough apart in memory to force the traversal of multiple routers, the delay of passing through several processing-routers can be simulated.

In both of these schemes, routing is different than in the basic memory. In the basic memory, access parcels always move from the root of the memory to the leaf and directly back to the root. In the bouncing threads execution model, the thread initiating the process enters from the root but then travels within the memory structure from leaves to routers and back to leaves without necessarily going back to the root. Routers in this scheme are more complex and multiple accesses (threads) may have paths that cross or even the same destination memory macro (figure 5.8).

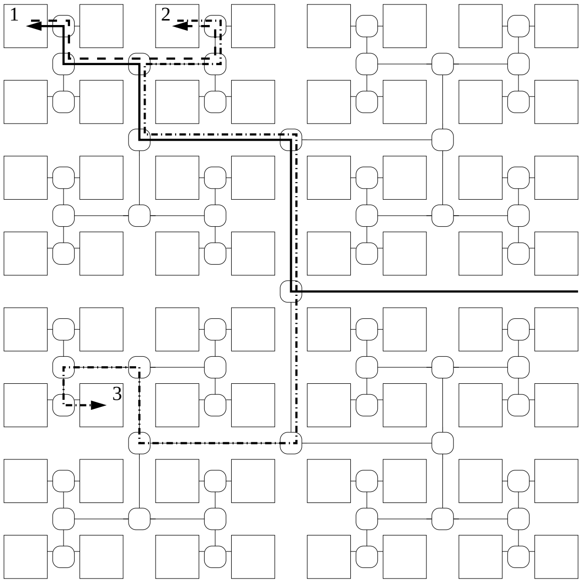


Figure 5.8. Threads travel between memory macros without returning to the root in the bouncing threads model. In this example, threads travel first to the memory macro marked “1”, then to “2” and finally to “3” without returning to the root.

Absolute addresses within the parcel are replaced by directions to match the routing method of memory accesses. Instruction addresses are replaced at iFetch, and data addresses are replaced at operand access time. In memory macros, then, addresses are stored as absolute addresses and are then converted to directions for the thread to travel to that destination. In effect, the directions scheme treats the latest origination point as the new root in the binary tree. As a result when using the processing in memory macro strategy, the routers still need to maintain no state about where they are in the H-structure, and they continue to operate in essentially the same was as described in the basic H-memory architecture relying on the leading edge of the thread. However, now either the memory macros need to know their address, or the thread must carry the address of the current memory macro (the destination it just arrived at) with it as additional state.

5.2.2 Simulator Description

The simulator is a cycle-accurate architecture that models the transitory latches in the simulated system. In a QCA design, the transitory latches in wires are the major source of delay and need to be explicitly modeled. The simulator acts on components of the microarchitecture as represented by modules. Each module can be as specific as a complete microarchitectural description or as general as a behavioral description with an appropriate wire delay included in the module. For instance, in the simulated system described below, the communication between modules is modeled by a wire module with every transitory latch explicitly represented (a complete description). The processor in the memory macro module, though, models the behavior of the processor and factors in the associated delay as a wire of appropriate length. In other words, the thread moves along a simple wire in the memory macro until it reaches the end of the module and the appropriate processor action is made in a single cycle update (i.e. an ALU operation). The length of the wire in the memory macro is used to model the delay that would be incurred by the thread if the entire processor circuitry were explicitly modeled.

The simulator can be either execution driven, trace driven, or pattern driven. This simulator is focused on exploration of the travel patterns and interaction of threads in the H-tree. For an execution driven system, an ISA can be defined in a module (in the memory macro module for the system simulated for this work). In trace mode, an input trace of addresses visited can be used. A pattern driven simulation could model thread travel given memory locality statistics. These three modes are interchangeable and allow different facets of the bouncing threads execution model to be explored.

5.2.3 Collision Detection and Avoidance

The frequency of collisions can have a substantial effect on the throughput and bandwidth of the system. The tradeoff is between having enough active threads to take advantage of the resources and having few enough threads so collisions are rare.

A collision occurs when more than one thread is trying to use the same router path. In the simulated system below, routing decisions are made by the router module and then passed to the collision handling module to determine if there is an impending collision and determine the appropriate action to avoid a collision. Each router has three directions of input: from the parent, from the right child, and from the left child. Each router also has three directions of output: to the parent, to the right child, and to the left child. A collision arises when more than one input path is trying to use the same output path. For instance, the thread coming from the left child and the thread from the right child are both trying to go to the parent (figure 5.9).

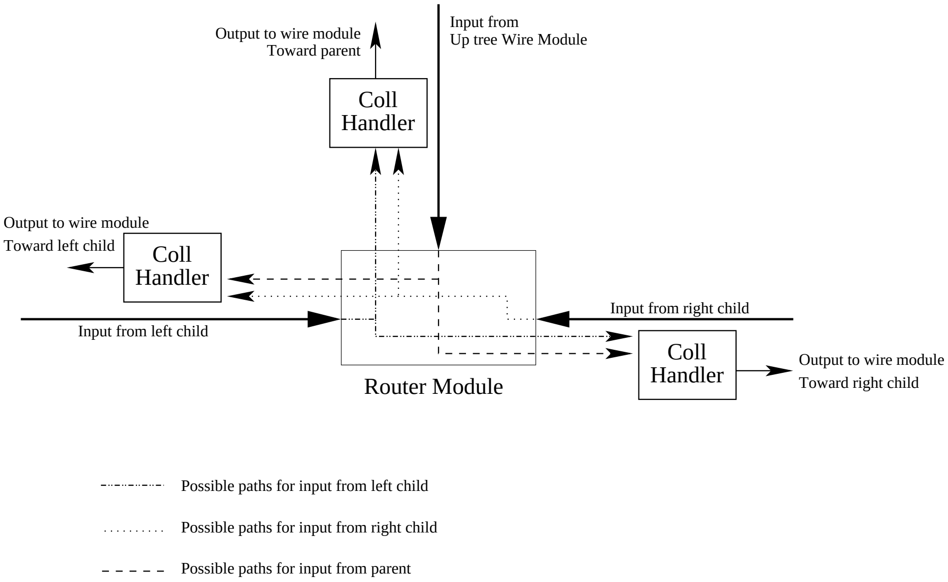


Figure 5.9. Collisions occur when two or more threads are trying to use the same output path at the same time.

5.2.4 Simulated System

In order to evaluate collision frequency and avoidance strategies, a simulator was written to allow execution of benchmarks. The simulator uses the processing in the memory macros strategy. The instruction set architecture is “Simple12”, an accumulator based ISA with a 4 bit opcode, 8 bit operand, and 12 bit word (table 5.2).

I/O Module

The I/O module controls the input to and the output from the H-structure. Initial commands are routed from here to begin simulations, and completed jobs are assumed to exit through the I/O module as well. If a command is waiting to be issued and the entrance to the H-structure is clear, the command is removed from the wait queue and inserted serially into the H (figure 5.12).

Table 5.2

Simple12 Instruction Set Architecture [4]

Opcode	Mnemonic	RTL
0000	JMP X	PC ⇐ X
0001	JN X	if A<0, PC ⇐ X else PC++
0010	JZ X	if A=0, PC ⇐ X else PC++
0100	LOAD X	A ⇐ M(X), PC++
0101	STORE X	M(X) ⇐ A, PC++
0110	LDI X	A ⇐ M(M(X)), PC++
0111	STI X	M(M(X)) ⇐ A, PC++
1000	AND X	A ⇐ A AND M(X), PC++
1001	OR X	A ⇐ A OR M(X), PC++
1010	ADD X	A ⇐ A + M(X), PC++
1011	SUB X	A ⇐ A - M(X), PC++
1111	END	

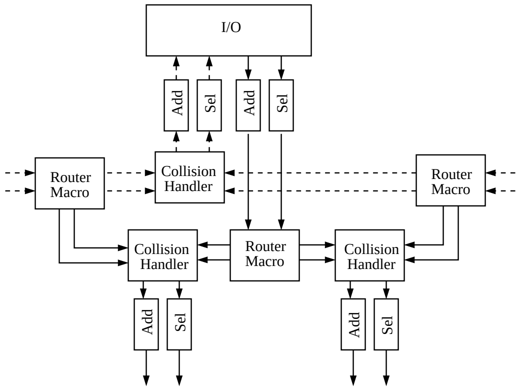


Figure 5.10. Simulator module interaction at I/O module and internal nodes.. Solid lines signify wire modules that represent down-tree connections. Dashed lines are wire modules that represent up-tree connections.

The output of the I/O module is two wires, address and select lines. The input depends on the collision strategy and is two or three wires: address and select lines and perhaps a collision handler line through which detoured threads previously

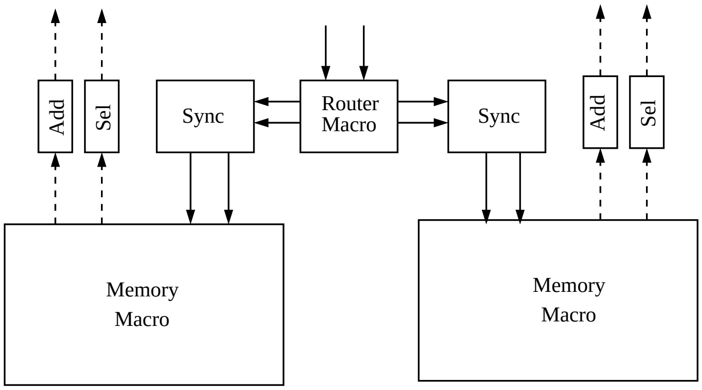


Figure 5.11. Simulator module interaction at memory macros. Solid lines signify wire modules that represent down-tree connections. Dashed lines are wire modules that represent up-tree connections.

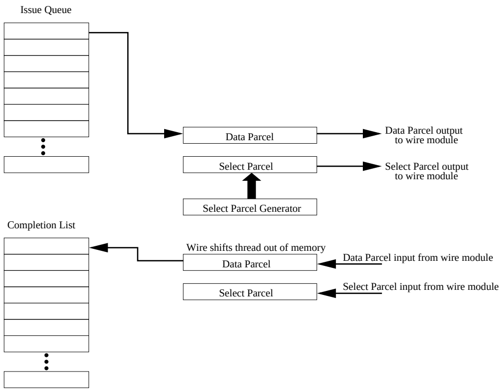

Issue Queue
Data Parcel
Data Parcel output to wire module
Select Parcel
Select Parcel output to wire module
Select Parcel Generator
Completion List
Wire shifts thread out of memory
Data Parcel
Data Parcel input from wire module
Select Parcel
Select Parcel input from wire module

Figure 5.12. I/O module handles the initial issue of a thread into the H-tree and the final notification of a process's completion.

involved in a collision are re-issued.

Collision Handler Module

The collision handler module is responsible for the detection of, avoidance of, and recovery from collisions which result from more multiple threads trying to use the same path. There are several potential implementation strategies. The common theme among them is the idea of a detour path for one of the colliding threads to take while the other thread uses the desired path. This detour can be a local route which is a wire that loops back to the same collision handler, or it can be a global path which takes the detoured thread to a collision handler at a different level in the tree (e.g. send the detoured thread up the tree to the parent of the current collision handler). The different collision handling strategies are a mix of the number and length of local detour routes and the number, length, and destination of global detour routes. The global routes can send threads horizontally up the tree, laterally to another subtree, or back to the root of the memory to be reinserted by the I/O module.

Global detour routes are more complicated than local routes because the directions of the parcel must be updated to reflect how to reach the desired memory macro from the new position.

Using detour routes increases the number of threads that can be involved in each collision (figure 5.13). In the basic set up, there can be two threads attempting to use the same path. With detour routes, though, there can be the initial two threads vying for the path as well as threads entering the collision handler from a detour path. For instance, if there is one local detour path, there could be as many as three threads involved in each collision.

In designing a collision handler, it is important to maintain a parity between the number of inputs to the collision handler and the number of outputs so that every thread that enters the collision handler has a path to follow if it cannot take its desired path immediately. It is not acceptable to “lose” a thread because it has no where to go.

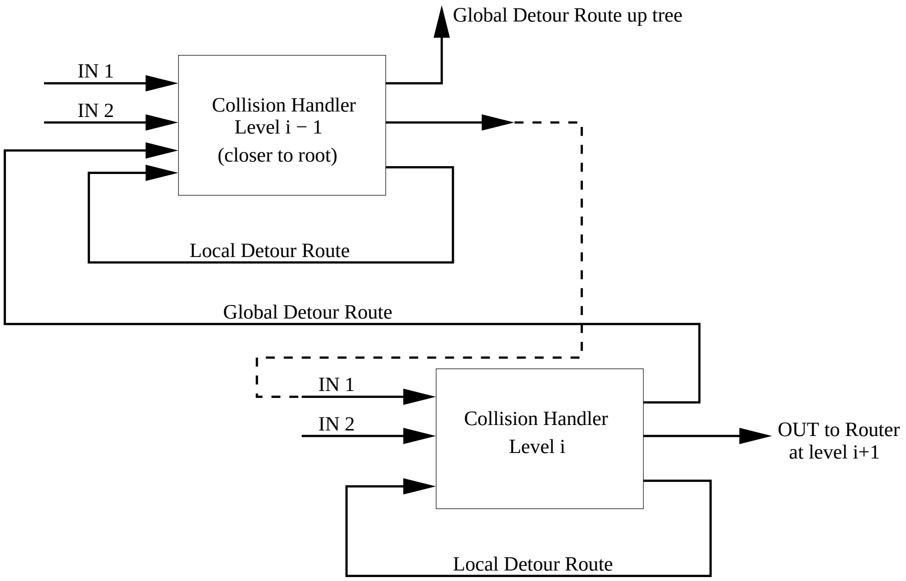


Figure 5.13. Creating detour routes in collision handlers increases the number of threads that can be involved in a single collision and increases the types of collisions that can occur. a) local detour route, b) global detour route

Wire Module

The wire module controls the downward wires (toward the memory macros) and the return wires (back up the tree toward the root). Wires are treated as shift registers. Each clock cycle, all input shifts forward by one position (figure 5.14).

Router Module

The router module directs threads to the next appropriate destination. Options are toward the nodes parent, left child, or right child. For the processing-in-router scheme, a fourth option is to the logic. The input consists of the address and select lines. Output is two wires to each possible destination (figure 5.15).

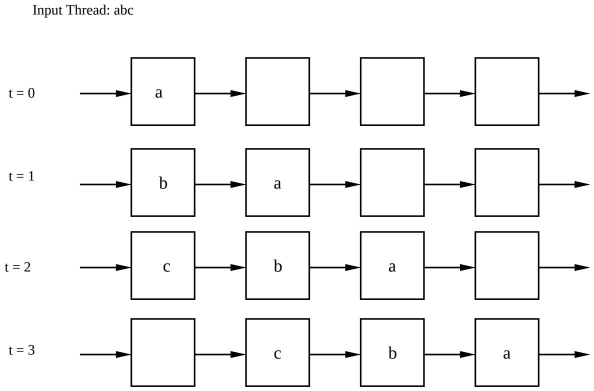


Figure 5.14. The wire module explicitly represents each transitory latch in the wire. Each update cycle, contents of the wire are shifted forward by one position. This wire spans a distance that requires four clock cycles to traverse.

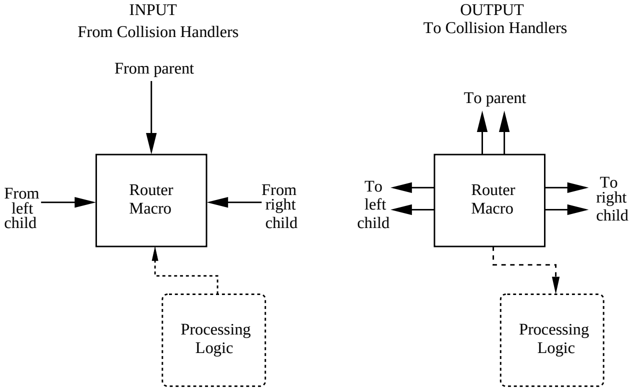


Figure 5.15. Router module is responsible for sending the thread toward the appropriate memory macro. It may or may not include processing logic.

Sync Module

A sync module sits in front of each memory macro to guarantee that incoming memory accesses interact with the stored data correctly (i.e. so the access begins with the start of the stored word's cycle). The sync module must also handle collisions in the case of a thread waiting to enter a memory macro and another thread entering the sync module to try to use the same memory location (figure 5.16).

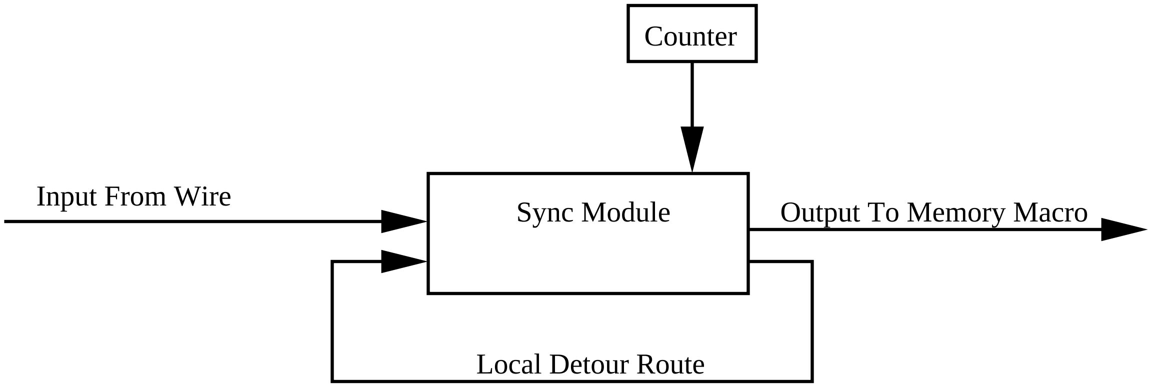


Figure 5.16. The sync module guarantees the thread enters the memory macro to correspond to the start of the word stored in the memory macro. The sync module also handles collisions that arise from multiple threads trying to access the same memory macro.

Memory Module

The memory module consists of the memory macro and for some implementations a simple processor. The input consists of the address and select wires coming from a sync module (figure 5.17).

5.2.5 Benchmark: Bubble Sort

In order to evaluate the architecture, bubble sort was executed as a benchmark. Bubble sort was chosen because it is complicated enough to include aspects of real programs such as loops and feedback but simple enough to be coded in the S12 ISA and executed in a reasonable amount of time. The instruction mix for a single thread executing the program is shown in table 5.3. The purpose of the simulation was to examine how the number of threads executing in the memory structure affects the

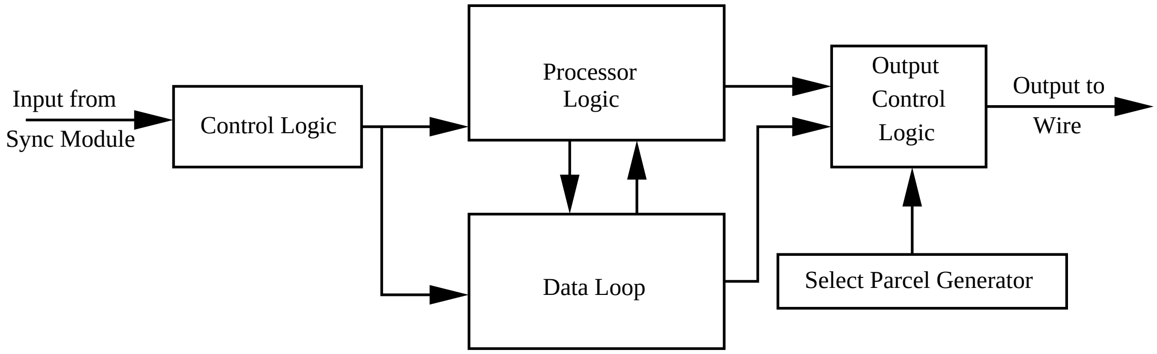


Figure 5.17. The memory module models the memory macros. Each memory macro has a word of storage and may include a serial processor.

threads and how the number of threads in the structure affects the use and role of collision handlers. To this end, four copies of the bubble sort program were placed in different address spaces. Threads are issued such that the first thread operates on the first copy of bubble sort. The second thread operates on the second copy in memory, and so on for the third and fourth threads issued. The fifth thread starts the pattern over and operates on the first copy in memory. In this way, the interaction between threads operating on shared memory addresses are modeled as well as threads operating on non-shared memory addresses.

For each copy of bubble sort in memory, the process was explicitly simulated assuming a serial processor at each memory macro. To generate the collision handler usage data for multiple threads, the trace mode of the simulator was used. In this way, every thread is acting as if it is executing the complete program (rather than sorting and resorting an actual list of integers in the memory).

5.2.6 Results

The goal of these simulations was to examine where collisions occur, how many threads are involved in each collision, and how the delay due to collisions effects the

Table 5.3

Bubble Sort Instruction Mix

Instruction	Count	Percent
jmp	27	6.9%
jn taken	12	3.1%
jn not taken	43	11.0%
jz taken	0	0%
jz not taken	0	0%
load	83	21.2%
store	54	13.8%
ldi	58	14.8%
sti	32	8.2%
and	0	0%
or	0	0%
add	27	6.9%
sub	55	14.1%
Total:	391	100%

execution time of the processes.

Where Collisions Occur

When considered by level in the memory structure, most collisions occur at the level closest to the memory macros (figures 5.18, 5.19, 5.20, 5.21). When normalized by the number of routers at each level, there are more collisions per collision handler at the top levels of the memory structure closest to the root (figure 5.22, 5.23, 5.24, 5.25). These figures are for 16 processes running concurrently in the memory structure, but the shape of the graphs is consistent regardless of the number of threads being executed. This suggests a fat tree structure may be useful to relieve contention close to the root of the memory structure.

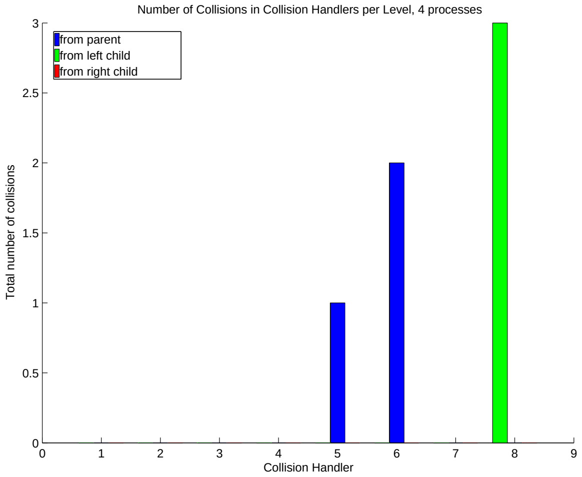


Figure 5.18. Total number of collisions at each level for 4 processes in the memory structure. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root with one collision handler.

Number and Size of Collisions

As the number of threads in the memory structure increases, the number of collisions first increases, and then drops dramatically (figure 5.26). However, the maximum

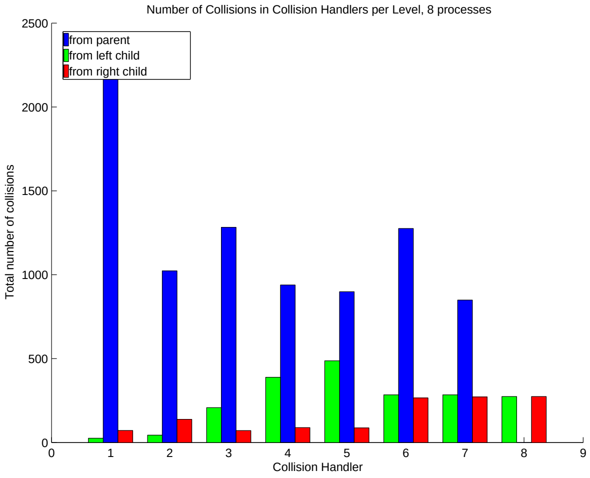


Figure 5.19. Total number of collisions at each level for 8 processes in the memory structure. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root with one collision handler.

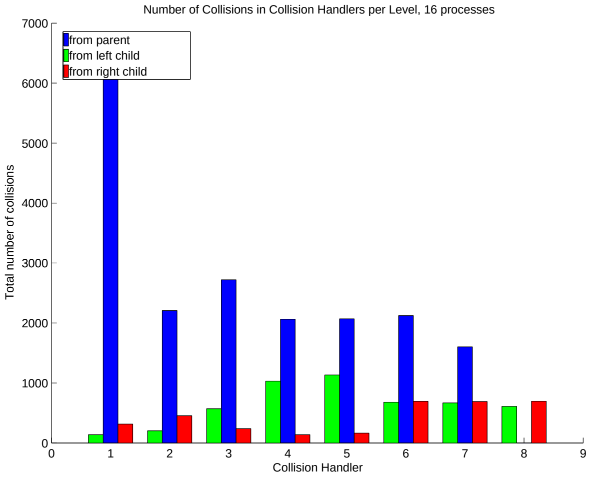


Figure 5.20. Total number of collisions at each level for 16 processes in the memory structure. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root with one collision handler.

number of threads involved is the number of threads executing. As the threads first collide at bottlenecks and are delayed, they are separated such that fewer collisions

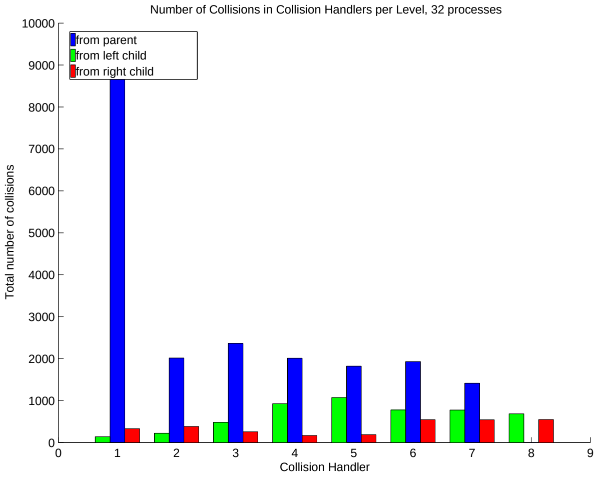


Figure 5.21. Total number of collisions at each level for 32 processes in the memory structure. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root with one collision handler.

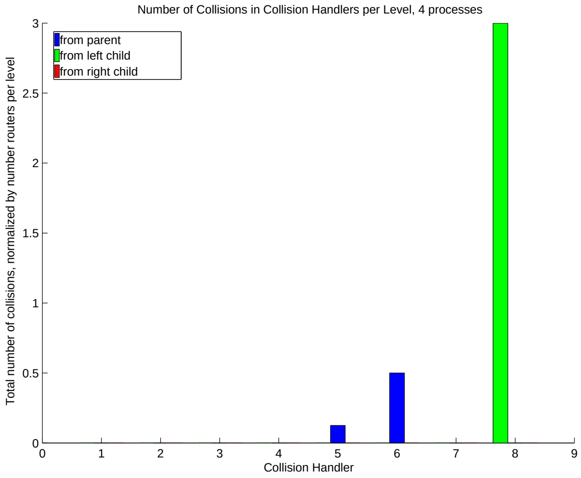


Figure 5.22. Number of collisions at each level for 4 processes in the memory structure normalized by the number of collision handlers at each level. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root and has one collision handler.

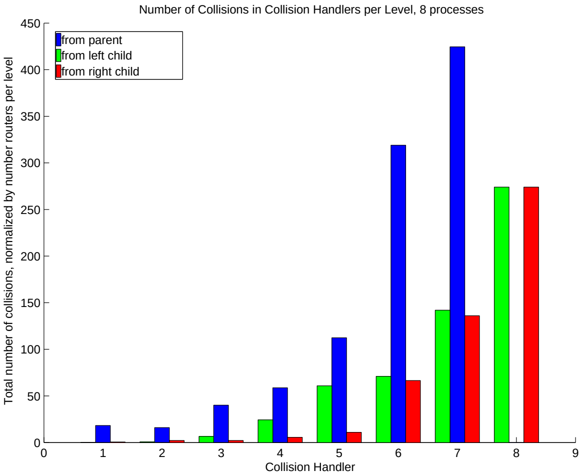


Figure 5.23. Number of collisions at each level for 8 processes in the memory structure normalized by the number of collision handlers at each level. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root and has one collision handler.

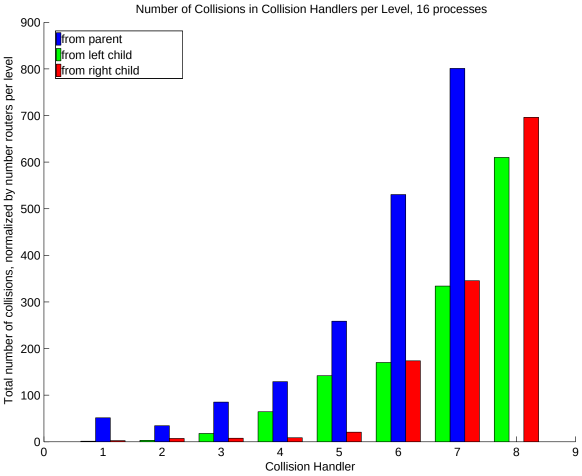


Figure 5.24. Number of collisions at each level for 16 processes in the memory structure normalized by the number of collision handlers at each level. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root and has one collision handler.

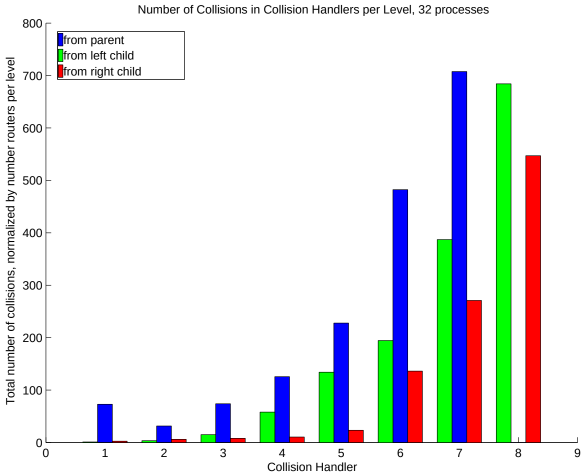


Figure 5.25. Number of collisions at each level for 32 processes in the memory structure normalized by the number of collision handlers at each level. Level 1 is closest to the memory macros and includes 128 collision handlers. Level 8 is at the root and has one collision handler.

occur overall. In other words, the necessary delay at the collision handlers acts as a built-in scheduling adjustment to better distribute the threads to incur fewer collisions in the future.

Execution Time

In a traditional processor, the argument for pipelining is that although an individual instruction may take longer than in a non-pipelined processor, the overall execution time for a process will be lower since the work of many instructions can be overlapped. The same holds true for this massively pipelined system. Although the total execution time for each thread is increased (figure 5.27), the average time over all processes executed is dramatically lowered (figure 5.28). The maximum number of cycles a thread is delayed increases with the number of threads being executed.

The execution time for a single thread executing alone is 123,881 cycles. For four threads executing at the same time, the final thread is delayed by 252 cycles.

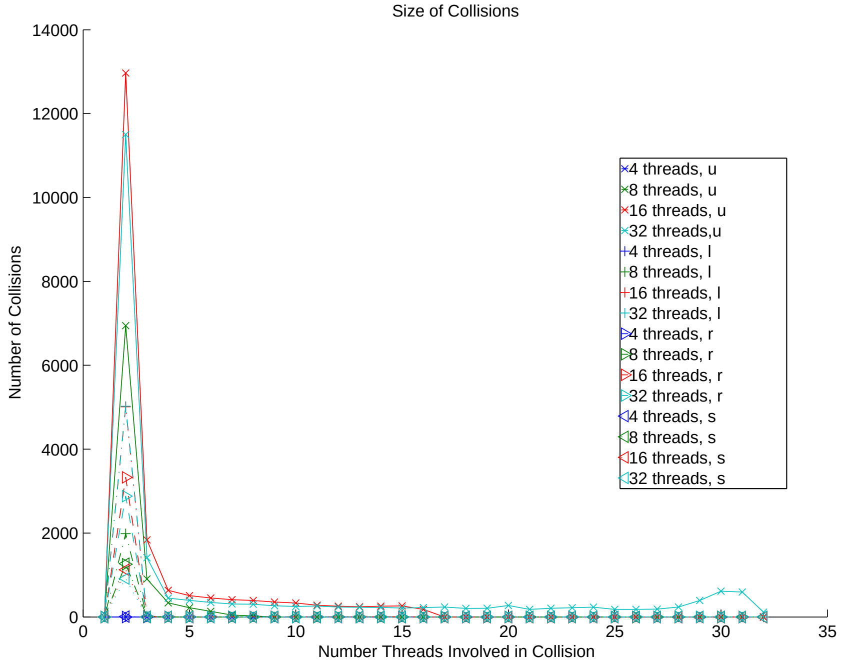


Figure 5.26. Frequency of collisions of each size depending on the number of processes in the memory structure. Regardless of the number of threads executing, most threads involve 3 threads or less. However, with n threads executing, there are a few collisions with as many as n threads involved.

However, for 32 threads executing concurrently, the last thread is delayed by 71,098 cycles. Despite this delay, the average execution time per process for 32 threads is 6093 cycles, as opposed to 31,033 cycles for four concurrent threads. This data affirms the value of pipelining and that its benefits are reaped in this massively pipelined system.

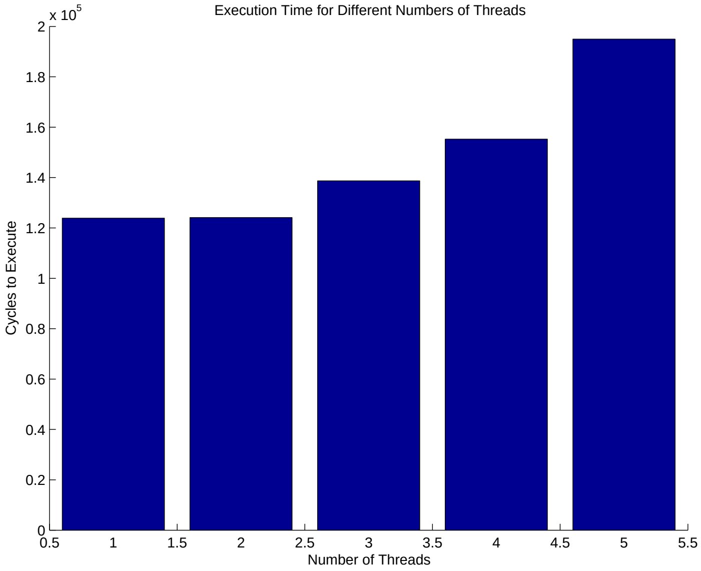


Figure 5.27. Execution time for all processes depending on how many processes are executing simultaneously.

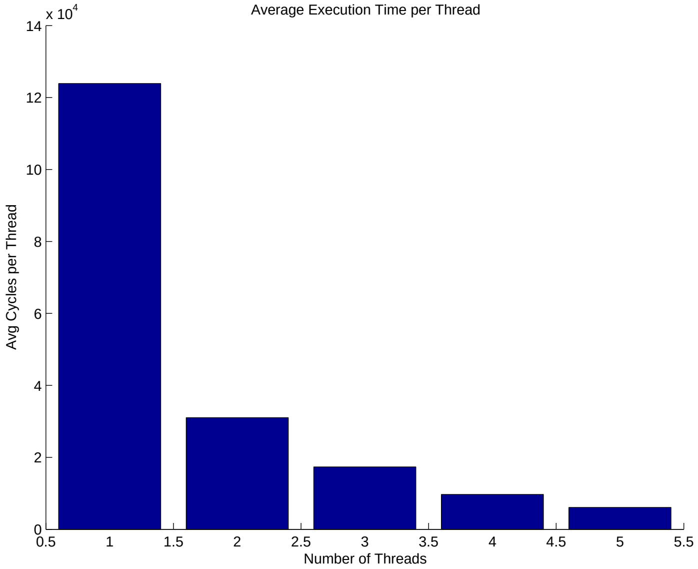


Figure 5.28. Average execution time for different numbers of processes executing simultaneously.

CHAPTER 6

Clocking Wire Layout Strategy

6.1 Introduction

The layout of the clocking wires will significantly impact the QCA circuit layout and resulting circuit density. The work discussed in earlier chapters assumed a minimum square clocking zone. While this is a useful tool until further implementation details are known because it allows one to capture some of the density limitations placed on the QCA circuits by the clocking wires, it is not a realistic model of clocking zones. More realistic clocking schemes will most likely not be based on static sized clocking zones, but on the computational wave scheme [10] in which the clocking wave travels across a QCA circuit and the switching occurs on the leading edge of the wave. In addition, previous work has assumed that the clocking wires will be a limiting factor on the density of QCA circuits. Leveraging others work on carbon nanotubes may allow the clocking wires to be eliminated as an impediment to QCA circuit density.

6.2 Clocking Wire Density

Carbon nanotubes (CNTs) were introduced in chapter 2 in the context of other non-silicon or nanotechnology memories. In addition to their use as non-volatile switches, metallic carbon nanotubes are also excellent conductors. Experimental and theoretical work with metallic single-walled carbon nanotubes (SWNTs) and multi-walled carbon nanotubes (MWNTs) suggest that the conductance of electrons is nearly ballistic. As a result, metallic CNTs should be excellent wires with low resistance and low heat dissipation.

In previous work, it was assumed that the clocking wires would be the limiting factor on the density of QCA circuits. However, the diameter of SWNTs is on par with the size of QCA molecules. A (10,10) SWNT is a conducting nanotube with a diameter of approximately 1.4 nm and lengths up to several hundreds of microns. This is the same size scale as the candidate QCA molecules being explored. Furthermore, there is no fundamental limit on how closely SWNTs can be placed. Even today, SWNTs can be placed precisely by nudging them with atomic force microscopes (AFMs). This is a very slow, laborious, error-prone process, but it indicates that the precise positioning of nanotubes is an engineering problem to be overcome rather than a fundamental limit of science.

The size of SWNTs and their potential pitches mean that the clocking wires do not have to limit the density of QCA circuits (figure 6.1). The linear clocking wire density ($wires/\mu m$) can be calculated by:

$$CDens = \frac{1}{max(p, CC_{dist})} \tag{6.1}$$

where p is the clocking wire pitch and CC_{dist} is the minimum distance between adjacent 4 dot QCA cells. The end of the roadmap, 2018, metal 1 pitch is 42 nm [28]. This implies a linear density of 23.8 $wires/\mu m$ for the end of the roadmap clocking wires. Using the (10,10) SWNTs, the limiting factor is the cell-cell distance of 2.8nm. The resulting linear density is 357.1 $wires/\mu m$. The linear density of the SWNT layout is 15 times greater than the end of the roadmap. This implies the SWNTs provide a potential overall density gain of 225 times over end of the roadmap metal wires.

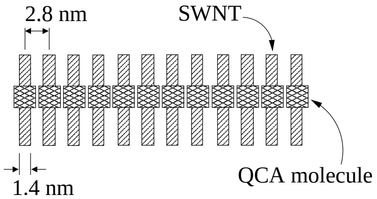


Figure 6.1. The size of SWNTs is on the same size scale as molecular QCA cells. Clocking wires made of SWNTs may not limit the density of QCA circuits. The diameter of the (10,10) SWNT is $1.4nm$. The center to center distance between QCA molecules is $2.8nm$.

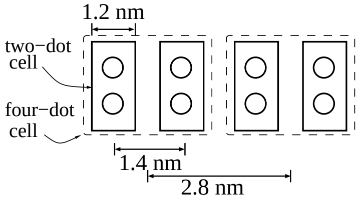


Figure 6.2. Cell to cell distance of a four dot QCA cell built from two 2-dot molecules.

6.3 Clocking Wire Layout

In the computational wave scheme, the clocking signal is generated by wires under the plane of the QCA molecules (figures 6.3, 6.4)[26]. The clocking wires generate a clocking field that supports QCA signal movement perpendicular to the clocking wires. Straight wires are very easy in this scheme, requiring only parallel clocking wires (figure 6.5a). Turning corners in this scheme is more difficult, requiring a range of clocking wire orientations to create the required clocking field (figure 6.5b). The result is that straight wires tend to be very area efficient while turns are inefficient.

An alternative to the perpendicular clocking wire layout scheme is to run the overlying QCA signals at a 45 degree angle to the clocking wires. This diagonal clocking wire layout strategy allows the QCA signal to travel along two directions on the same set of clocking wires (figure 6.5c).

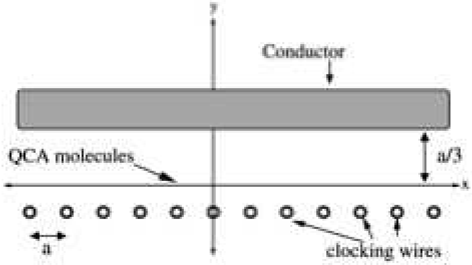


Figure 6.3. Schematic of clocking wires below the plane of a QCA circuit. Clocking wires run perpendicularly out of the viewing surface. QCA cells are placed on the plane sandwiched between the clocking wires and an upper conducting plate. [26]

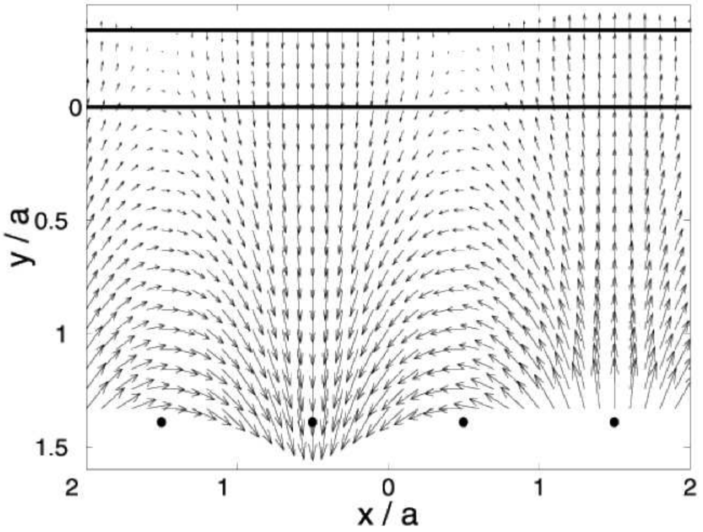


Figure 6.4. Electric field generated by CMOS wires placed under the plane of the QCA molecules. Block dots are the CMOS wires which run perpendicular to the viewing plane. The layer where QCA molecules reside is located at the horizontal 0 line. The highest black line is the conductor as seen in figure 6.3. [26]

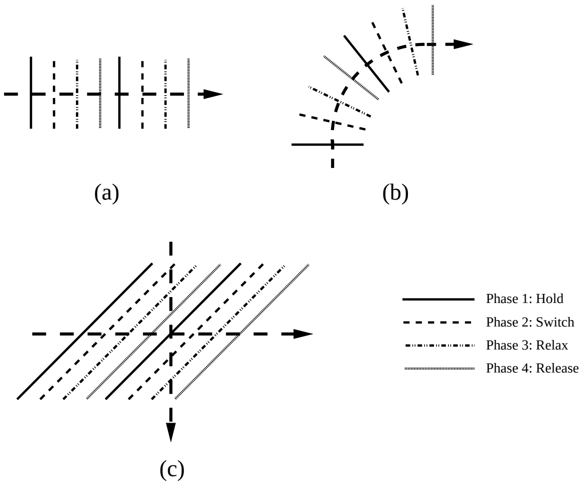


Figure 6.5. Three clocking wire layouts: (a) Straight QCA signal traveling perpendicular to the clocking wires. (b) QCA turn with perpendicular wires. (c) QCA signal traveling at 45 degree angle to clocking wires.

6.4 Diagonal Clocking Scheme

The diagonal clocking scheme requires two wire orientations (positive slope and negative slope) and two orderings (dictates direction of travel) to allow travel to the north, south, east, and west. The two orientations required are perpendicular to each other. In one orientation the clocking wires have a positive slope, and in the other orientation clocking wires have a negative slope. The two wire orderings determine which direction the clocking signal travels (i.e. the relative phases between neighboring wires). The result of this is four different combinations of orientation and ordering, or four "meta-zones." Each meta-zone is a clocking wire layout that allows travel in two directions. The meta-zones and their supported directions of travel are shown in figure 6.6.

Each meta-zone, then, supports two directions of movement and four possible types of movement. For instance, meta-zone I allows straight movement to the east, straight movement to the south, a turn from east to south, and a turn from south to east.

A spiral requires four meta-zones to loop back onto itself. Using the labels in figure 6.6, a spiral with data moving clockwise, starting in the upper left corner, requires a meta-zone that supports motion north and east (IId), east and south (Ic), south and west (IVd), and west and north (IIIc).

QCA signals must be able to cross from one meta-zone to another. This can be done only at locations where the clocking zone on each side of the meta-zone border is the same (figure 6.7). These locations are referred to as "channels." In figure 6.7, meta-zones II and I are shown abutted. When laid out in this manner, channels occur every other clocking wire. The location of these channels is determined by the layout of the clocking wires and their relative phase. The location of channels is static assuming the relative phases of the clocking wires does not change, as is assumed throughout this work.

6.5 Evaluating Clocking Strategies

In evaluating clocking strategies, there are two areas to consider. The first is the resulting density of the QCA circuits. The second is the level of difficulty to manufacture the given layout. For all nanoelectronics, making connections between wires and manufacturing arbitrary shapes will most likely be very difficult. As one considers clocking strategies then, it is important to keep in mind the eventual manufacturing challenges that can be anticipated.

The memory loop used in the H-memory is a good circuit on which to evaluate the clocking strategies. First, since it is a regular structure with feedback, the circuit lends itself to comparison across clocking strategies. Second, the data loop is a major component of the H-memory and analyzing the effect of the clocking strategies on

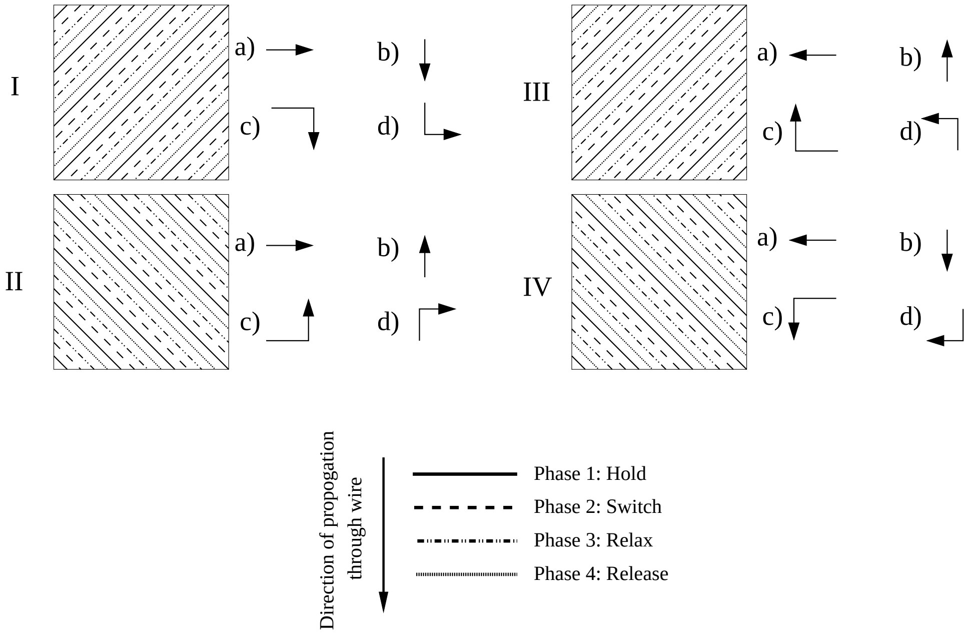


Figure 6.6. I-IV Diagonal wire layout metazones. a-d) Supported direction of movement of overlying QCA signals.

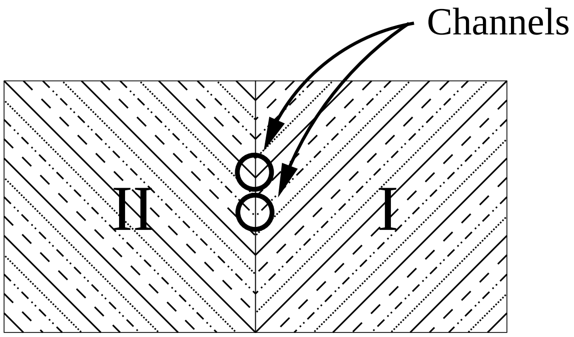


Figure 6.7. Channels between two meta-zones. For the four phase clock scheme, channels occur every other clocking wire along the border between two meta-zones.

the data loop provides important insight into how the clocking strategies would impact the overall density of the H-memory.

The density of the circuit can be captured by calculating the number of bits stored in each data loop and the area required. Insight into the manufacturability of the data loops can be gained by calculating the number of clocking wires required to implement the data loop.

There are a number of ways to layout the data loop. Three are considered here to give an idea how the two clocking layout strategies perform across a range of layouts. The first configuration is the standard spiral used in the implementation of the H-memory. The second configuration is the flattened spiral that takes advantage of efficient straight stretches of QCA wires. The third is the snake that is inefficient in terms of use of clocking wires but can lead to high densities in some circumstances.

The snake configuration consists of a "leg" of straight wire then a turn and a returning leg of straight wire parallel to the first leg but traveling in the opposite direction. Neighboring legs are traveling in opposite directions. This requires distinct sets of clocking wires for each leg. So, while with the legs of the spirals move in the same and allow a single set of clocking wires to be shared among all legs, the snake configuration requires a distinct set of clocking wires for each leg (as seen in the perpendicular layout in figure 6.8).

An example of the clocking wire layout needed to implement each of these in the perpendicular scheme can be seen in figure 6.8 and in the diagonal scheme in figure 6.9.

The perpendicular scheme is very efficient for long stretches of straight QCA signal. However, when the QCA signal needs to turn, it is less space efficient because the clocking wires cannot be placed as closely as they can in the straight QCA signal. Since the clocking wire needs to maintain its orientation perpendicular to the direction of travel of the QCA signal, a broad range of wire orientations is needed. On the inside of the turn, the distance between the clocking wires is the clocking wire pitch. On the outside of the turn, the distance between the clocking

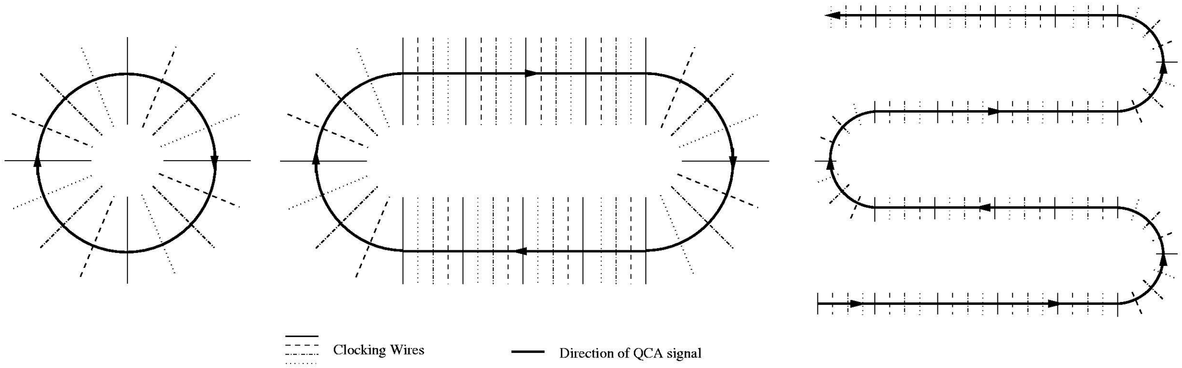


Figure 6.8. Three loop configurations with the perpendicular clocking wire layout strategy. The configurations are the standard spiral, the flattened spiral, and the snake configuration. The bold line shows the direction of the overlying QCA signal. The patterned lines indicate the relative phase of each wire.

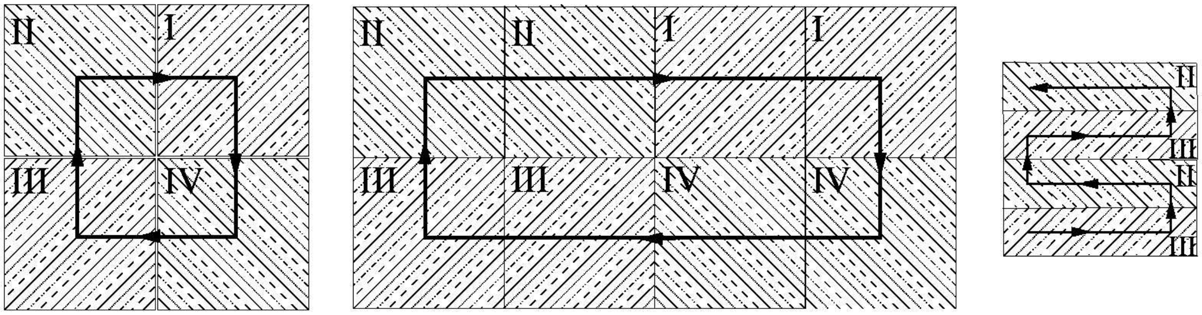


Figure 6.9. Three loop configurations with the diagonal clocking wire layout strategy. The configurations are the standard spiral, the flattened spiral, and the snake configuration. The bold line shows the direction of the overlying QCA signal. The patterned lines indicate the relative phase of each wire.

wires depends on the radius of the turn and the number of clocking wires used to make the turn. These parameters and others will be further discussed below (figure

6.10). Because clocking wires on the turn cannot be packed as closely as the clocking wires along the straight stretch, the turns are the limiting factor on the potential density of the QCA circuits designed for the perpendicular clocking scheme.

The parameters that determine the density of the QCA circuit designed for the perpendicular clocking scheme on the turn include the clocking wire pitch (p), the maximum separation between clocking wires that will still properly transmit the QCA signal (m), the half angle between clocking wires (α), and the length of the clocking wires (w) (figure 6.10). From these, the active radius of the turn, or where QCA circuits can be expected to function properly (x), and the overall turn radius (r) can be calculated. Finally, in order to calculate the overall densities the QCA wire pitch (Q), the number of turns in the spiral or snake (n), and the length of the extension (l) for the flattened spiral and snake configurations are needed.

The parameters needed to calculate the circuit density for the diagonal clocking scheme include the clocking wire pitch (p), the channel distance (c), the number of turns in the spiral or snake (n), and the length of the extended side (l) for the flattened spiral and snake configurations.

The equations to calculate the number of bits stored in a given data loop and the area it requires can be found in table 6.1. The equations for the number of wires required for each configuration can be found in table 6.2.

The resulting density for a sample set of loops can be seen in figure 6.11. For the perpendicular data loops, the number of wires in a complete circle was held steady at 16. The maximum separation between the ends of the clocking wires around turns was ignored. However, one can see that as the number of turns increases for these data loops, the density decreases rapidly. For data loops with many loops in the spiral, using more clocking wires to make a smoother turn is more efficient. For few loops in the spiral, a turn made with fewer clocking wires is more efficient. This

Table 6.1

Number of Bits, Area Equations for two clocking schemes

Layout type	Bits Stored	Area
Diag spiral	$2n(n-1)$	$32p^2(n-1)^2$
Diag flat	$n(n-1)(\frac{n}{2})(\frac{l}{2\sqrt{2}p(n-1)}+4)$	$8p^2(n-1)^2(\frac{l}{2\sqrt{2}p(n-1)}+4)$
Diag snake	$n(2+\frac{l}{c})$	$2nc^2(2+\frac{l}{c})$
Perp spiral	$\frac{\pi n}{4\alpha}$	πr^2
Perp flat	$\frac{n}{2}(\frac{\pi}{2\alpha}+\frac{l}{p})$	$\pi r^2 + 2lr$
Perp snake	$n(\frac{\pi}{4\alpha}+\frac{l}{2p})$	$n(\pi {r_1}^2 + 2lr_1)$

Table 6.2

Number of Wires Used for two clocking schemes

Layout type	Number Wires
Diag spiral	$16(n-1)$
Diag flat	$4(n-1)(\frac{l}{2\sqrt{2}p(n-1)}+4)$
Diag snake	$2n(2+\frac{l}{c})$
Perp spiral	$\frac{\pi}{\alpha}$
Perp flat	$\frac{\pi}{\alpha}+\frac{2l}{p}$
Perp snake	$n(\frac{\pi}{\alpha}+\frac{2l}{p})$

Table 6.3

Data Loop Density Parameters

Parameter	Value	Comment
c-c dist	$2.0nm$	cell-cell distance
QCA pitch	$4.0nm$	$2 *$ cell-cell distance
α	$\frac{\pi}{16}$	Angle between perpendicular scheme clocking wires on the turn (figure 6.10)
p	$2.8nm$	SWNT pitch
m	$10nm$	maximum clocking wire separation to continue to transmit QCA signal
b	variable	calculated as in table 6.1
r Perp Flat	$(\frac{p}{2} + \frac{b * num_{loops}}{\pi} / \sin\alpha)$	radius of perpendicular turns
c	$4 * \sqrt{2}$	channel distance, 2*SWNT pitch * $\sqrt{2}$
l perp	$44.8nm$	16 wires * p, used for flattened spiral and snake configurations
l diag	$31.7nm$	4 metazones * length of metazone side

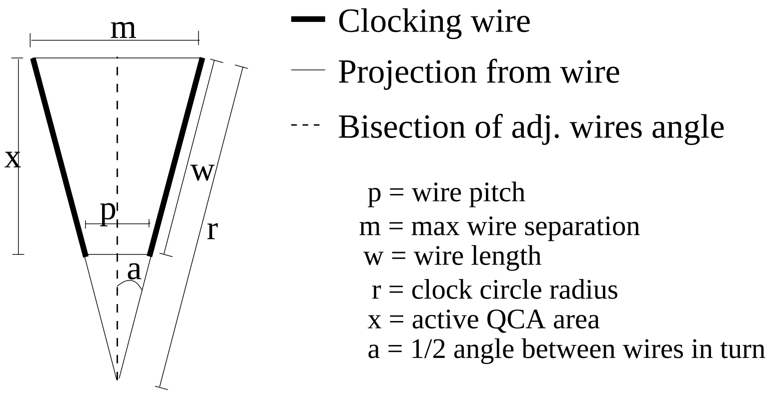


Figure 6.10. Parameters used to calculate clocking wire density on a turn for the perpendicular clocking wire layout.

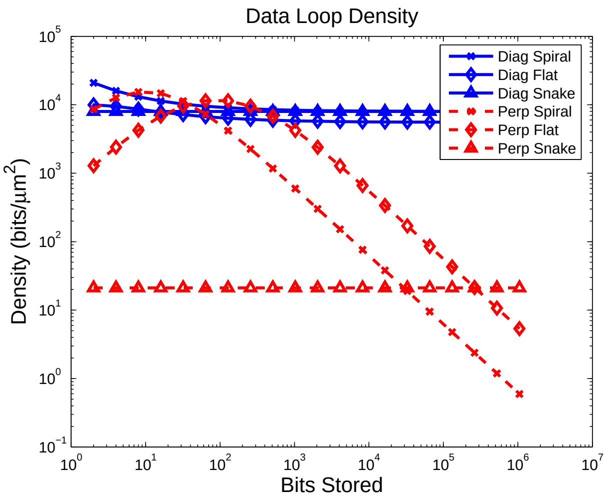


Figure 6.11. Density of data loops using the perpendicular clocking wire layout scheme and the diagonal clocking wire layout scheme for different QCA wire widths. The clocking wire pitch is limited by the size of the QCA pitch. Uses parameters in table 3.

is because fewer loops in the spiral require less active area from the clocking wires (a smaller x in figure 6.10) allowing the angle of the clocking wires to be larger than if a bigger active area was required.

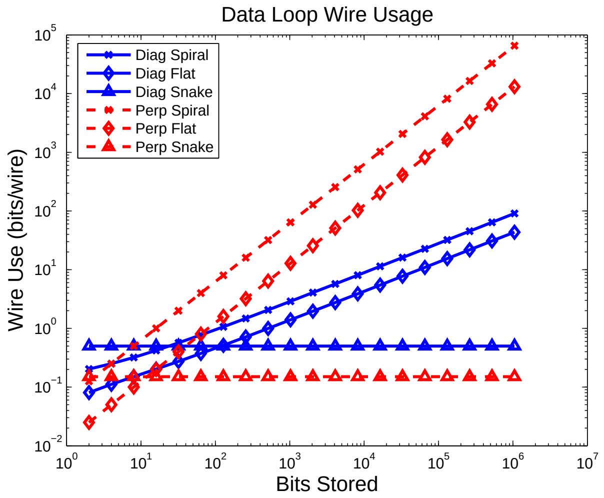


Figure 6.12. Clocking wire usage by data loops using the perpendicular clocking wire layout scheme and the diagonal clocking wire layout scheme for different QCA wire widths.

The snake configuration designed for the perpendicular clocking wire scheme has a steady density because since it requires only a single QCA signal to be supported through turns, the turning radius can be very small and nearly as efficient as the clocking scheme is for straight signals.

The density of the diagonal configurations changes very little. This is because the meta-zones are equally dense for turns as for straight signals. The density of the diagonal layouts is competitive with the best density of the perpendicular layouts and configurations and far exceeds the average density of the perpendicular layouts.

The number of bits compared to the number of wires used is an interesting number. The higher the bits/wire measure, the more clocking wires are reused, indicating a more efficient use of clocking wires. It also indicates that fewer individual clocking wires need to be fabricated. Since fabrication at the nanoscale is expected to be more error prone than fabrication at the microscale, requiring fewer individual features is an important feature. This bits/wire measure, then, gives a rough indication of the relative manufacturability of the clocking wire layout compared to the density of the QCA circuit (figure 6.12).

The perpendicular spiral and flattened spiral give the best wire usage figures for large numbers of bits stored. However, these configurations and layouts also have the worst storage density. The wire usage of the diagonal spiral and flattened spiral are the next best. They achieve a good balance between circuit density and wire usage. In addition, the diagonal layout requires clocking wires in only two orientations that are at right angles to each other. These features along with the versatility of the clocking wire scheme indicate its utility and importance.

6.6 Conclusion

The use of SWNTs for clocking wires removes the clocking wires from the QCA circuit density equation. In addition, the diagonal clocking wire layout scheme allows higher density layouts and more easily manufactured clocking wire circuitry, as well as more versatile clocking wire layouts. The diagonal clocking layout strategy also opens up new possibilities for clocking floor planning work. With robust and versatile clocking floor plans that are manufacturable, QCA circuits can be designed to these floor plans, speeding the potential fabrication of QCA circuits.

CHAPTER 7

Conclusions and Future Work

Quantum-dot cellular automata is a novel device with characteristics that differ substantially from CMOS. This work presented a new memory architecture native to QCA that takes advantage of the characteristics of the device to achieve memory densities that exceed those of end of the roadmap CMOS.

The transistor paradigm has been very successful at the microscale. However, as devices continue to shrink, the short comings of the transistor paradigm become exacerbated. The barriers to extending transistors into the nanoscale are physical, economic, and architectural in nature. QCA offers solutions to all of these concerns. Transistors today operate in spite of quantum effects that dominate at the nanoscale. QCA makes explicit use of these effects. As transistors continue to shrink, the leakage current begins to dominate the circuit which leads to increased heat dissipation. The heat problem is intensified because of the desire to take advantage of the small device size and make circuits as dense as possible. This leads to unworkable heat densities. QCA offers an avenue toward very high density with low power requirements and minimal heat dissipation. The economic barriers of chip fabrication costs can be addressed by taking advantage of the self-assembling nature of the molecular world. Finally, the architectural challenges of using nanoscale transistors that are more susceptible to permanent and transient faults has been addressed in this thesis and by the prior QCA logic design work discussed in chapter 1. In short, QCA is poised to breach the red brick wall of the ITRS roadmap.

This work is the first memory architecture designed in QCA that looks beyond the array memory paradigm that has worked so well in CMOS but which is not suitable for QCA. By designing with an eye explicitly on the device characteristics, these devices can be exploited to create efficient designs. The H-memory implementation discussed in this work is by no means the most efficient design possible, and yet it is still able to obtain orders of magnitude gains in density over the end of the roadmap projections for DRAM and SRAM densities. The first cut design discussed in chapter 4 achieves projected densities of hundreds of gigabits per square centimeter as opposed to end of the roadmap DRAM density of 58.21 $Gbits/cm^2$. The weakness of the H-memory is the long latencies required due to the shift-register nature of wires. Access times are measured in the thousands of clock cycles. However, despite the long latencies, the maximum bandwidth of the H-memory still far exceeds that of end of the roadmap DRAM because of the H-memory's fine grained pipelining and a clock rate in the tens of terahertz.

Future circuit designs of the H-memory should address the white space in the current design within the memory and routing macros as well as between macros. By eliminating or making use of this space, the major component of latency, wire delay, can be improved.

In addition to a basic memory architecture, the novel bouncing threads execution model was further explored. This execution model eliminates the von Neumann bottleneck and addresses the long latencies of memory access time by moving the processing into the memory structure itself. Rather than moving data from the memory to the processor and back again, the process state is moved to the data. This allows massive multi-threading and takes advantage of the fine-grained pipelining in the memory structure. The bouncing threads execution model is a fertile area for future research. ISA functionality, pipeline stage placement within the memory structure, the utility vs cost of instruction caches with the threads, the design of collision handlers that are guaranteed to handle all incoming threads, and the expansion of the H-tree into a fat tree at congested levels are all topics to be explored.

Finally, a clocking wire layout strategy was proposed that eliminates the clocking wires as a limit to circuit density by proposing the use of single-walled carbon nanotubes for clocking wires. In addition, the diagonal clocking wire layout scheme was proposed that allows floorplans which are more flexible and space efficient than those explored via the perpendicular clocking scheme.

BIBLIOGRAPHY

[1] $http://www.chem.ucla.edu/dept/faculty/stoddart/new/graphics/blue_{t}imeline.gif$.

[2] *http : //www.ipt.arc.nasa.gov/carbonnano.html*.

[3] *http : //www.mb.tn.tudelft.nl/nanotubes.html*.

[4] Introduction to simple12 assembly and rtl. Dept. of Computer Science and Engineering, University of Notre Dame (2004), Introduces S12 ISA and RTL in undergraduate computer architectour course(CSE 321).

[5] *Quantum-dot Cellular Automata: beyond transistors to extreme supercomputing* (October 2004).

[6] D. Antonelli, T. Dysart, D. Chen, X. Hu, A. Kahng, P. M. Kogge, R. C. Murphy and M. Niemier, Quantum dot cellular automata (qca) circuit partitioning: Problem modeling and solutions. In *41st Design Automation Conference (DAC)* (June 2004).

[7] R. Bennewitz, J. Crain, A. Kirakosian, J.-L. Lin, J. McChesney, D. Petrovykh and F. Himpsel, Atomic scale memory at a silicon surface. *Nanotechnology*, 13: 499–502 (2002).

[8] G. H. Bernstein, I. Amlani, A. Orlov, C. Lent and G. Snider, Observation of sitching in quantum-dot cellular automata cell. *Nanotechnology*, (10): 166–173 (1999).

[9] D. Berzon and T. Fountain, Computer memory structures using qca. Technical report, University College London (1998).

[10] E. P. Blair, *Tools for the Design and Simulation of Clocked Molecular Quantum-Dot Cellular Automata Circuits*. Master's thesis, University of Notre Dame (2003).

[11] M. Brehob, R. Enbody, Y.-K. Kwon and D. Tomanek, The potential of carbon-based memory systems. *IEEE*, pages 110–114 (1999).

[12] J. B. Brockman and P. M. Kogge, The case for processing-in-memory. Technical report, University of Notre Dame (1997).

[13] Y. Chen, D. A. A. Ohlberg, X. Li, D. R. Stewart and R. S. Williams, Nanoscale molecular-switch devices fabricated by imprint lithography. *Applied Physics Letters*, 82(10): 1610–1612 (March 2003).

[14] A. DeHon, Array-based architecture for molecular electronics. In *First Workshop on Non-silicon Computation* (February 2002).

[15] M. Dresselhaus, G. Dresselhaus, P. Eklund and R. Saito, Carbon nanotubes. Physicsweb, $http : //physicsweb.org/box/world/11/1/9/world-11-1-9-1$.

[16] T. Dysart and P. M. Kogge, Strategy and prototype tool for doing fault modeling in a nano-technology. In *IEEE Nano Conference* (August 2003).

[17] M. B. Elowitz and S. Leibler, A synthetic oscillatory network of transcriptional regulators. *Nature*, 403: 335–338 (January 2000).

[18] S. E. Frost, A. F. Rodrigues, C. A. Giefer and P. M. Kogge, Bouncing threads: Merging a new execution model into a nanotechnology memory. In A. Smailagic and N. Ranganathan, editors, *IEEE Computer Society Annual Symposium on VLSI: New Trends and Technologies for VLSI Systems Design*, pages 19–25, IEEE Computer Society (February 2003).

[19] S. E. Frost, A. F. Rodrigues, A. W. Janiszewski, R. T. Rausch and P. M. Kogge, Memory in motion: A study of storage structures in qca. In *1st Workshop on Non-Silicon Computation (NSC-1), held in conjunction with 8th Int. Symp. on High Performance Computer Architecture (HPCA-8), Boston, MS* (Feb 3 2002).

[20] C. Giefer, S24 datasheet. Research directed by Arun Rodrigues and P.M. Kogge at the Univ. of Notre Dame.

[21] C. Giefer, Simulation data. Research directed by Arun Rodrigues and P.M. Kogge at the Univ. of Notre Dame.

[22] C. Giefer, Simulation results. Research directed by Arun Rodrigues and P.M. Kogge at the Univ. of Notre Dame.

[23] S. C. Goldstein and D. Rosewater, Digital logic using molecular electronics. In *ISSCC* (February 2002).

[24] L. K. Grover, A fast quantum mechanical algorithm for database search. In *28th Annual ACM Symp. on the Theory of Computing*, pages 212–219 (1996).

[25] J. R. Heath, P. J. Kuekes, G. S. Snider and R. S. Williams, A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280: 1716–1721 (June 1998).

[26] K. Hennessy and C. S. Lent, Clocking of molecular quantum-dot cellular automata. *J. Vac. Sci. Technol. B*, 19(5): 1752–1755 (Sep/Oct 2001).

[27] Y. Huang, X. Duan, Y. cui, L. J. Lauhon, K.-H. Kim and C. M. Lieber, Logic gates and computation from assembled nanowire building blocks. *Science*, 294: 1313–1317 (9 November 2001).

[28] ITRS, International technology roadmap for semiconductors 2000 update. Technical report, ITRS (2000).

[29] ITRS, International technology roadmap for semiconductors 2003 update. Technical report, ITRS (2003).

[30] B. Keeth and R. J. Baker, *DRAM Cricuit Design: A Tutorial*. Series on Microelectronic Systems, IEEE Press (2001).

[31] P. M. Kogge, Qca memory presentations.

[32] Y.-K. Kwon, D. Tomanek and S. Iijima, "bucky shuttle" memory device: Synthetic approach and molecular dynamics simulations. *Physical Review Letters*, 82(7): 1470–1473 (February 1999).

[33] C. Lent, P. D. Tougaw and W. Porod, Quantum cellular automata: The physics of computing with arrays of quantum dot molecules. *Proceedings Workshop on Physics and Computation*, pages 5–13 (1994).

[34] C. S. Lent and B. Isaksen, Clocked molecular quantum-dot cellular automata. *IEEE Trans. on Electron Devices*, 50(9): 1890–1896 (September 2003).

[35] Z. Li and T. P. Fehlner, Molecular qca cells. 2. characterization of an unsymmetrical dinuclear mixed-valence complex bound to a au surface by an organic linker. *Inorganic Chemistry*, 42(18): 5715–5721 (2003).

[36] M. Lieberman, S. Chellamma, B. Varughese, Y. Wang, C. Lent, G. Bernstein, G. Snider and F. Peiris, Quantum-dot cellular automata at a molecular scale. *Ann. N.Y. Acad. Sci.*, (960): 225–239 (2002).

[37] G. Moore, Cramming more components onto integrated circuits. *Electronics*, 38(8) (April 19 1965).

[38] D. P. Nackashi and P. D. Franzon, Molectronics: A circuit design perspective. In *International Conference on SPIE Smart Electronics and MEMS, Melbourne, Australia*, volume 4263, pages 80–88 (2000).

[39] M. Niemier, *Designing Digital Systems in Quantum Cellular Automata*. Master's thesis, University of Notre Dame (April 2000).

[40] M. Niemier, *The Effects of a New Technology on the Design, Organization, and Architectures on Computing Systems*. Ph.D. thesis, University of Notre Dame (September 2003).

[41] M. Niemier and P. M. Kogge, Designing complex logic systems with qca devices. In *Great Lakes Symposium of VLSI* (March 1999).

[42] M. Niemier and P. M. Kogge, Logic-in-wire: Using quantum dots to implement really dense processing logic. In *Proceedings of the Third Petaflops Workshop, with Frontiers of Massively Parallel Processing* (February 1999).

[43] M. Niemier and P. M. Kogge, Logic-inwire: Using quantum dots to implement a microprocessor. In *International Conference on Electronics, Circuits, and Systems (ICECS '99)* (September 1999).

[44] M. Niemier and P. M. Kogge, The 4-diamond circuit: A minimally complex nanoscale computational building block in qca. In *IEEE Symp. on VLSI (ISVLSI)*, pages 3–10 (February 2004).

[45] M. Niemier, A. Rodrigues and P. M. Kogge, A potentially implementable fpga for quantum dot cellular automata. In *1st Workshop on Non-Silicon Computation (NSC-1), with 8th Intl. Symposium on High Performance Computer Architecture (HPCA-8)* (February 2002).

[46] M. T. Niemier and P. M. Kogge, Exploring and exploiting wire-level pipelining in emerging technologies. In *International Symposium of Computer Architecture*, pages 166–177, ISCA 2001, Sweden (July 2001).

[47] M. T. Niemier and P. M. Kogge, Problems in designing with qcas: Layout = timing. *Int. J. of Circuit Theory and Applications*, 29: 49–62 (4 January 2001).

[48] A. O. Orlov, I. Amlani, G. Toth, C. S. Lent, G. H. Bernstein and G. L. Snider, Experimental demonstration of a binary wire for quantum-dot cellular automata. *Applied Physics Letters*, 74(19): 2875–2877 (May 1999).

[49] A. R. Pease, J. O. Jeppesen, J. F. Stoddart, Y. Luo, C. P. Collier and J. R. Heath, Switching devices based on interlocked molecules. *Acc. Chem. Res.*, 34(6): 433–444 (2001).

[50] B. Prince, *Semiconductor Memories: A Handbook of Design, Manufacture, and Application*. John Wiley & Sons, second edition edition (1991).

[51] B. Prince, *High Performance Memories: New Architecture DRAMs and SRAMs - evolution and function*. John Wiley & Sons (1996).

[52] B. Prince, *Emerging Memories: Technologies and Trends*. Kluwer Academic Publishers (2002).

[53] A. Rodrigues, Bouncing threads execution model back-up slides.

[54] T. Rueckes, K. Kim, E. Joselevich, G. Y. Tseng, C.-L. Cheung and C. M. Lieber, Carbon nanotube-based nonvolatile random access memory for molecular computing. *Science*, 289: 94–97 (7 July 2000).

[55] P. Rutten, M. Tauman, H. Bar-Lev and A. Sonnino, Is moore's law infinite? the economics of moore's law. In *Kellog TechVenture 2001 Anthology*.

[56] P. W. Shor, Algorithms for quantum computation: Discrete logarithms and factoring. In *IEEE Symposium on Foundations of Computer Science*, pages 124–134 (1994).

[57] S. Thompson, M. Alavi, M. Hussein, P. Jacob, C. Kenyon, P. Moon, M. Prince, S. Sivakumar, S. Tyagi and M. Bohr, 130 nm logic technology featuring 60nm transistors, low-k dielectrics, and cu interconnects. *Intel Technology Journal*, 6(2): 5–13 (May 16 2002).

[58] P. Tougaw and C. Lent, Logical devices implemented using quantum cellular automata. *Journal of Applied Physics*, 75 (1994).

[59] L. M. Vandersypen *et al.*, Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, (414): 883–887 (December 2001).

[60] K. Wallus, G. Schulhof, T. Dysart, A. Vetteth, G. A. Jullien, V. S. Dimitrov and J. Eskritt, $http://www.atips.ca/projects/qcadesigner$.

[61] Y. Wang and M. Lieberman, Thermodynamic behavior of molecular-scale quantum-dot cellular automata (qca) wires and logic device. *IEEE Transactions on Nanotechnology*, 3(3): 368–376 (September 2004).

[62] R. Weiss, S. Basu, S. Hooshangi, A. Kalmbach, D. Karig, R. Mehreja and I. Netravali, Genetic circuit building blocks for cellular computation, communications, and signal processing. *Natural Computing, an International Journal* (2003).

[63] K. Yano, T. Ishii, T. Sano, T. Mine, F. Murai, T. Hashimoto, T. Kobayashi, T. Kure and K. Seki, Single-electron memory for gitga-to-tera bit storage. *Proceedings of the IEEE*, 87(4): 633–651 (April 1999).