Molecules, Gates, Circuits, Computers

Seth Goldstein and Mihai Budiu

October 10, 2003

Contents

1 Introduction

The last 40 years have witnessed persistent exponential growth in computer performance. This incredible rate of growth is a phenomenon known as Moore's law [Moo65]. Gordon Moore predicated in 1965 that the density of devices on a memory chip would double every 18 months. His prediction has proven true. Moore's law has also come to mean that processing power per dollar doubles every year. The most incredible result of this success is that we have come to count on it. In fact, we expect our computers, video games, and cell phones to get smaller, lighter, more powerful, and cheaper every year.

Advances in semiconductor fabrication have brought smaller devices and wires, increased speed and chip density, greater reliability, and decreased cost per component. This remarkable success is based largely on complementary metal-oxide semiconductor (CMOS) integrated circuits. Improvements to CMOS technology–along with periodic, but less predictable, developments in computer architecture–have essentially cut the cost of processor performance in half every 18 months.

Today we can begin to see the limits of the growth predicted in Moore's law. The physics of deep-submicron CMOS devices, the costs of fabrication, and the diminishing returns in computer architecture all pose important challenges. The strengths of CMOS have been that its transistors and wires have almost ideal electrical characteristics and that the individual components are manufactured simultaneously with the end product. In other words, when a computer chip is under construction, its transistors and wires are manufactured on the final chip in their final location. This differentiates semiconductors from all other complex manufactured products and is responsible for their low cost. However, as the size of individual components shrinks below 100 nm, their behavior is no longer close to ideal and fabrication costs begin to soar. Further, computer architecture, which constrains the performance of silicon, is reaching limits of its own.

Chemically assembled electronic nanotechnology (CAEN) is a promising alternative to CMOS-based computing that has been under intense investigation for several years. CAEN uses self-alignment to construct electronic circuits out of nanometer-scale devices that take advantage of quantum-mechanical effects. In this chapter we explore how CAEN can be harnessed to create useful computational devices with more than 10^{10} gate-equivalents per cm^2. The strategy we describe substitutes compilation time (which is inexpensive) for manufacturing precision (which is expensive). This can be achieved through a combination of reconfigurable computing, defect tolerance, architectural abstractions, and compiler technology. The result is a high-density, low-power substrate that will have lower fabrication costs than its CMOS counterparts.

Using electronic nanotechnology to build computers requires new ways of thinking about architecture and compilation. CAEN, unlike CMOS, is not practical for constructing complex aperiodic structures. Thus CAEN-friendly architectures are based on dense regular structures that can be programmed after fabrication to implement complex functions. For example, nanoBlocks [GB01],

nanoCells [HHP+01], cellular automata [LDK01], quantum cellular automata cells [NRK02], and small PLAs [DeH02] are all programmable structures that can be built out of nanoscale components. Each of these combines nanoBlocks (or similar structures) to form a computational fabric, such as a nanoFabric [GB01], that can be altered after manufacture.

Because CAEN-based devices have a higher defect density than CMOS devices, they require built-in defect tolerance. A natural method of handling defects is to design the nanoFabric (or similar device) for self-diagnosis and then implement the desired functionality by configuring around the defects. Reconfigurability is therefore integral to the operation of any nanoFabric. Fortunately, many of the nanoscale components used to build nanoFabrics are well suited for reconfigurable computing.

In reconfigurable computing, the functions of programmable logic elements and their connections to storage are changed during operation to create efficient, highly parallel processing kernels tailored for the application under execution. The network of processing elements is called a *reconfigurable fabric*. The data used to program the interconnect and processing elements are called a *configuration*. Examples of current reconfigurable fabrics include commercial field programmable gate arrays, for example [Xil02, Alt02], and research prototypes such as Chimaera [YMHB00] and PipeRench [GSM+99]. As we show later, one advantage of nanoFabrics over CMOS-based reconfigurable fabrics is that the area overhead for supporting reconfiguration is virtually eliminated. This magnifies the benefits of reconfigurable computing, yielding computing devices that may outperform traditional ones by orders of magnitude in many metrics, such as computing elements per cm^2 and operations per watt.

In this chapter we take the reader from an informal definition of a bit to the architecture of nanoFabrics. We begin with an overview of some important features of any computing-device technology. We analyze the evidence in favor of digital (as opposed to analog) computation. We then discuss the benefits of using electrons as the basis of information exchange. Finally, we describe the features digital logic must have in order to manage the complexity inherent in computers with hundreds of millions of individual components.

After describing the fundamental implementation technology and its salient characteristics, we provide, in Section 3, an introduction to computer architecture and its recent history, including an explanation of how computers work and the methods by which designers create reliable systems. This overview explains what CAEN-based devices must do to compete with CMOS devices. We conclude the first half of the chapter, in Section 5, with a description of reconfigurable computing.

The second half of the chapter explores CAEN-based computing in greater depth. We begin, in Section 6, by describing the devices and fabrication methods available to the computer architect. These tools constrain the possible architectures that can be built economically. For example, one constraint around which to design an architecture is to disallow three-terminal devices. This simplifies fabrication but could reduce the efficiency of the resulting chip because three-terminal devices appear essential in the construction of restoring logic families.

In Section 8.3 we describe some circuit families that could be supported with molecular electronics. In particular, we describe how a restoring logic family can be constructed from a molecular latch. The latch is built using only two-terminal devices and clever fabrication techniques. This latch provides some of the benefits of three-terminal devices without requiring a processing technique that can colocate three different wires in space. In Section 9 we describe the general space of molecular architectures and in Section 9.5 we describe some of the constraints imposed on any molecular architecture. In Section 9.1 we give, as a detailed example, a complete nanoFabric architecture.

While we focus on computers, nearly everything discussed applies to other electronic devices. In fact, a computer can be considered a generalization of all electronic circuits because it is a programmable device that can perform the task of any circuit, though perhaps in a slower or less efficient manner. If we ignore the amount of power, time, and cost required, a programmable computer can offer the same functionality as any electronic circuit.

Thus, the ideas presented here are equally applicable to general-purpose computers, cell phone controllers, video game consoles, thermostats, radio receivers, and countless other devices.

2 The Switch

In this section we explore the basic building block of a digital computer. We begin with an abstract definition of information to show that using digital circuits—and more specifically—binary digital circuits, is not arbitrary but rather makes such circuits more robust and less error sensitive. We then describe the necessary features that logical operations require to support the design and implementation of complex digital circuits. We go on to show how logic gates can be realized with transistors. Using the transistor as an example, we conclude the section with a discussion of the essential characteristics that a fundamental building block must have to support digital logic.

2.1 Information

In this section we discuss the two fundamental components of a computer: switches and wires. Switches are used to manipulate information and wires are used to move information around. To meaningfully discuss the switches and wires that comprise a computer, we must establish a context for their use. Thus, we define what a computer is and what it needs to do. Computers are complex devices that serve a variety of functions. Thus, there are numerous different, but essentially equivalent, definitions of computers. For our purposes we define a computer as a device that manipulates information. The reason for this will become clear.

Other sources also define a computer as an information manipulation device. For example, *Merriam-Webster* defines a computer as "a programmable electronic device that can store, retrieve, and process data" [MW00]. The *Encyclopaedia Britannica* likewise defines a computer as "any of various automatic electronic devices that solve problems by processing data according to a prescribed sequence of instructions" [Enc02]. Both definitions are unnecessarily limited because they require the computer to be electronic; we show below that this is not a *sine qua non* of computers. However, both definitions agree that a computer is a device that "processes data" or, as we put it earlier, a device that manipulates information.

Our definition of a computer begs two additional questions: what is information and what does it mean to manipulate information? Information, as an entity that can be measured, is a relatively modern concept introduced by Shannon in 1948 [Sha48]. To have information about a system is to be able to distinguish something about the system. For example, if I have a two-faced coin on a table, it can either be heads up or tails up. If I tell you the coin is heads up, then you can distinguish between the two possible states of the system. If the system were, instead, a room with a light and I told you the light was on, then you would again be able to distinguish between the two possible states of the system, light and dark. The amount of information in each of these systems is the same: heads/tails for the coin, on/off for the light.

Suppose instead that there are two coins on the table—a nickel and a penny to be precise. To distinguish the state of the entire system you would need to know whether both coins are both heads up, whether both are tails up, whether the penny was heads up and the nickel was tails up, or vice versa. In this system, there are four states. Similarly, if I have a light bulb that can be off, or in one of three levels of brightness, then to describe the state of the light bulb would require choosing between one of four states (off, dim, normal, bright). As the number of states in a system increases, the amount of information needed to describe it also increases.

The smallest amount of information is the amount needed to distinguish between one of two states. (If there is only one possible state to the system, then one cannot distinguish anything about it. Therefore there is no information in describing it. For example, when you ask a colleague how they are and they reply, "too busy," you get no information since they are always too busy.) We call this amount of information a bit.[1] One bit of information is enough to

[1] Bit is shorthand for binary digit. A binary digit is a digit in base two (i.e., a one or a zero).

distinguish between one of two states: heads or tails, on or off, zero or one.

We now follow a common rule in computer engineering: whenever possible build a system using the simplest possible set of complete components. By a "set of complete components," we mean a set of components from which all the desired functionality can be obtained. For example, in this case, we want to be able to manipulate any amount of information. Since the smallest amount of information is a bit, can we just manipulate bits? If so, then we have a very simple system indeed. In fact, strings of binary digits can be used as the form of information that will be manipulated in a computer.

Thus, to keep the basic devices in a computer as simple as possible we limit them to working on bits of information, that is, they operate on binary digits. If we need to distinguish between more than two states we can use strings of binary digits. Two bits of information are enough to distinguish between one of four states. In general, $\log_2 n$ bits of information is enough to distinguish between one of n states. A string of n bits can be used to distinguish 1 out of 2^n numbers, apples, bank accounts, etc. In other words, a string of n bits can be used to represent any number from 0 to $2^n - 1$. Of course, we can use numbers to represent the letters in the alphabet and strings of numbers can then be used to represent words, etc.

In a computer, everything is represented by strings of binary digits. The strings of binary digits can be interpreted as numbers, letters, dollars, or apples. In the end, everything represented in the computer is simply information. This is our first encounter with abstraction, arguably the central concept of computer science. Thus, whether the computer is manipulating your bank account, simulating the universe, or displaying a video image, it is just manipulating information. To make this manipulation as easy as possible, the information is represented as a string of binary digits.

Another reason for using binary digits (and not decimal digits, for example) is robustness. Informally, robustness is increased because it is easier to distinguish between two things (e.g., a light is either on or off) than between tens or hundreds of things (e.g., if the light is on a dimmer it may be one-third on or two-tenths on, etc.). Thus, we will require the basic elements of our computer only to be capable of manipulating binary information.

Information is an abstract notion but computers are concrete entities that must manipulate concrete entities. In other words, the abstract notion of a bit must be implemented with a real world bit. What attributes should this concrete bit have? It should always be in one of two states. When in one state it should stay there until actively, but easily, changed. It should be easy to distinguish between the two states of the bit. It should be inexpensive. Finally, the implementation of the bit should be separate from the meaning of the bit. This last requirement is another example of abstraction. It may be the fundamental abstraction of computer science and the reason that Moore's law is possible [Ben02]. This requirement means that the information carried by a bit should not depend on the information carrier. Whether a bit is implemented as the position of a relay (involving billions of atoms) or as the spin of a single electron, the information content is the same. It is either a "1" or a "0." Or, in the context of Moore's law and electronic chips, it means that the devices on a chip can shrink and voltage levels can be scaled down. Yet it is still possible to implement a bit (i.e., it is still possible to distinguish between one of two states: a logical one and a logical zero).

Many different information carriers have been proposed: mechanical rods [Dre86], magnetic moments [ES82], amino acids [RWB+96], proteins [WB02], voltage levels, etc. Voltage levels are the traditional means for representing information in computers. Thus, the definitions of computers include the word "electronic." Essentially a digital computer uses two voltage levels to represent the two different values of a bit. One level, ground, represents a zero, and the other, V_{dd}, represents a one. The main reason voltage levels (which are just the potential energy levels of electrons) are so useful is that voltages can be changed quickly and with very little energy. This is because electrons are light and fast. However, it is hard to evaluate more thoroughly how effective an information carrier is until we also examine how we store and manipulate the information.

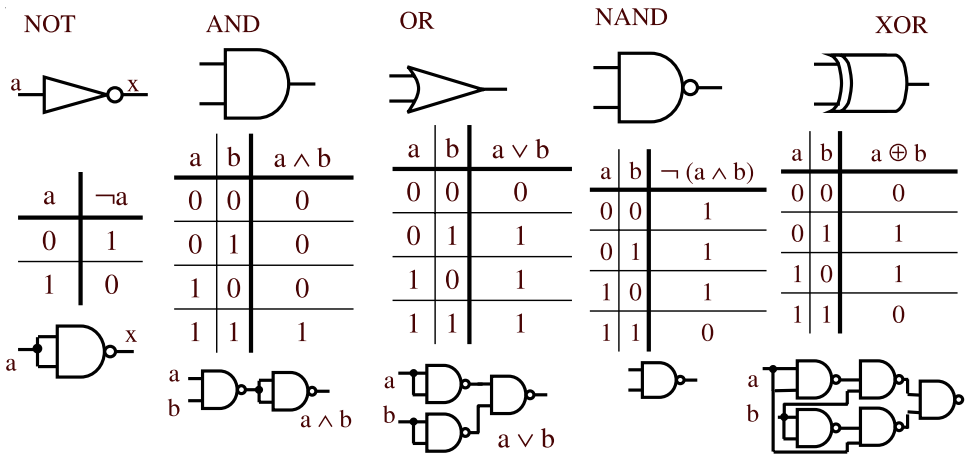


Figure 1: Some Boolean logic gates, their symbols, and how they can be implemented using a NAND gate. At the top of each column is the logical symbol for the operation. Under that is its corresponding truth table. Below that is an implementation of the truth table using only NAND gates.

2.2 Boolean Logic Gates

For the reasons noted above, computers manipulate information using operators that act on binary values. We call such operators Boolean logic gates. Boolean logic was first studied by George Boole who showed that all arithmetic computations can be synthesized from a very small number of logical primitives, such as logical AND and NOT. These two primitives are a complete set, meaning that they are sufficient to implement all logical functions. Fig. 1 shows how some common logic gates can be implemented using just the logical NAND gate, which is an AND gate followed by a NOT gate.

While it is possible to construct any logical function out of NAND gates, it is certainly tiresome to do so. For example, Fig. 2 shows the implementation of a one bit half-adder using NAND gates. A half-adder adds two binary digits and produces the sum and carry of the addition as output. However, by comparing the truth table, or the gate implementations, we can see that a simpler representation is available. The carry function can be implemented with a two-input AND gate and the sum function with a two-input XOR gate. Using an AND and XOR gate instead of a collection of NAND gates is a way of managing complexity. It is certainly easier to confirm that the implementation of the half-adder is correct by looking at two gates, the AND and XOR

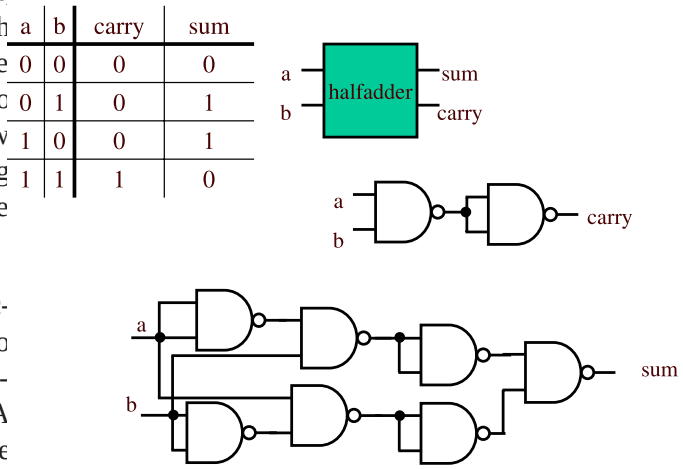


Figure 2: An implementation of a half-adder using NAND gates.

gates, than looking at nine. This is another example of using the power of abstraction. In this case, we only need to know the behavior of the XOR gate to verify that sum is implemented correctly. It is not necessary to know the implementation of it, since we know how it behaves.

However, for abstraction to work, we must be able to ignore the implementation and focus only on the behavior of a gate. Furthermore, for abstraction to be truly useful, we need to be able to predict the behavior of the gate in many different and possibly complex circumstances. For example, Fig. 3, shows three different and functionally equivalent representations of a 1-bit full-adder. The ability to abstract is the key to managing complexity. Since modern computers require millions of gates, this is one requirement for the design and implementation of computers. This ability imposes certain requirements on the implementation of the actual logic functions and the information carriers.

The first requirement is that a logic gate needs to ensure that its output can be used as the input of another logic gate. The information produced must be recognizably either a zero or a one by the input of the next gate. Furthermore, the wire along which the information is transmitted must also not destroy the information. In a perfect world, each bit of information produced by a gate or carried by a wire would be either exactly at the level of a logic zero (ground or zero volts in typical electronic systems) or a logic one (V_{dd}). In practice, various environmental factors combine to degrade the voltage levels so that what should be at zero volts is instead close to zero, but not exactly zero. If we are to robustly perform calculations in the presence of the noise introduced by the environment, we must be able to recognize nearly zero values as zero and nearly one values as one. Further, we want to produce as output a value that is no worse, and hopefully better (i.e., closer to its perfect value), than the input. In other words, our elementary computing devices should be able to restore nearly perfect values back to perfect values. The more ambiguous the input is allowed to be before giving an incorrect answer, the more robust our system will be. If each gate acts to restore the values closer to their ideal values, we can compose many of them together into a complex system without concern over where in the circuit the gate is.

By examining the full-adder implementation it is also evident that the output of a logic gate may have to go to more than one input of another gate. For example the ha-sum output (the output of the gate labeled “x”) of the “a+b” half-adder feeds three NAND gates in the second half-adder. Furthermore, we do not know how the gates will be arranged physically, so we cannot know how long the wires are between the gates. This means that the gates must be able to drive many inputs (at least two from our example) and that the gates must drive long or short wires. The former notion is called fan-out. Any implementation of logical gates must have a fan-out of at least two (i.e., allow a gate to drive at least two other gates).

The final requirement we list here is that the inputs and outputs of a logic gate must be isolated from each other. This requirement ensures that the output of a logic gate is a function of its inputs and not the other way around. Isolation of inputs and outputs allows a circuit to be designed without concern about how the output will be used. If, on the other hand, the output could influence the input, then the designer would have to know, in detail, how the output might change. Otherwise, the output might change the input that would cause changes to ripple throughout the system. This is an essential feature if one intends to build circuits with more than one level of logic.

If the logic gates fulfill these three requirements (restoring logic, input/output (I/O) isolation, and fan-out), a computer designer can create new logical operations from previously designed operations without regard to how the previously designed operations were implemented. We gloss over the fact that the logic operations do not have infinite fan-out. In other words, there is a contract in the abstraction that says how many devices a particular operator can drive. The designer must stay within that contract. Of course, if we have fan-out of two NAND gates, then we can construct gates with any fan-out by building a tree of NAND gates, where each level of the tree can drive twice as many gates as the previous one. An automatic tool may do this for a designer. Likewise, an automatic tool may analyze the contract for the logic gates and ensure that wires being driven by the gates

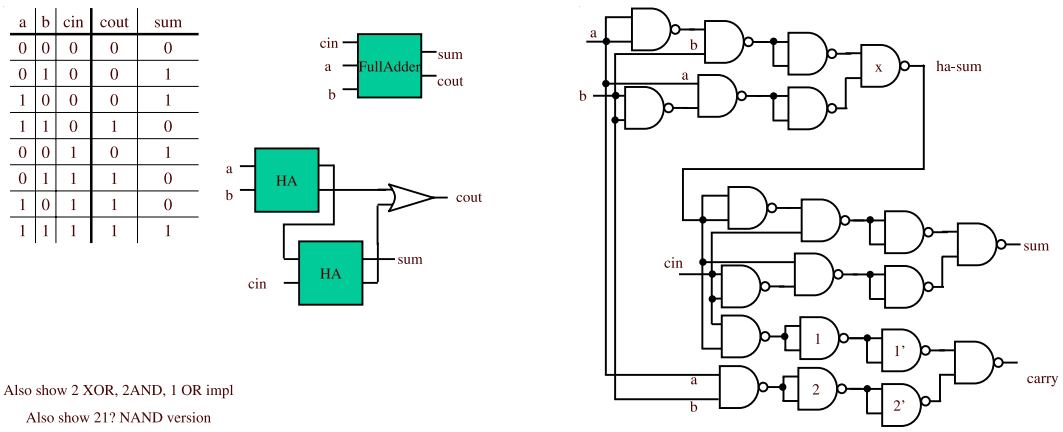


Figure 3: Several different representations of a full-adder.

never exceed the contracted length. Thus, the tools the designer uses help to support the abstraction. In fact, the complexity of modern computer design requires tools to support the abstractions.

When designing a family of logical operations one will want to ensure that they fulfill at least these requirements. However, one will probably not want to add too many other requirements to the contract. As we will show later in Section 4, adding to the contract may make it easier for the user of the component, but often at a substantial cost. In other words, one cost of abstraction is that the contract can be more general than necessary. For example, if we required that logic gates support a fan-out of 20, then every instance of a logic gate would have to support such a fan-out, even if most of them only needed to drive one or two other gates. Another example of how abstraction can incur overhead is present in the design of the full-adder in Fig. 3. The version built from half-adders will include extra gates because the designer is not allowed to peek into the half-adder abstraction. The simplest example of this is that the inverter gates (1, 1′, 2, and 2′ in the figure) at the output of the carry out logic and the input of the OR gate cancel out and are not necessary. This is an example of how allowing optimization across an abstraction layer can provide benefits.

So far we have only described combinational logic. Any change on the inputs flows through the circuit and appears at the outputs. However, even the simplest of circuits often need some kind of storage or synchronization.[2] When the storage elements are small and part of the circuit they are often called registers or latches. Registers allow information to be saved over time and also allow different computations to synchronize together.

2.3 Transistors

There have been many robust implementation vehicles for binary logic, but nothing has compared to the transistor in terms of its overall economy, reliability, speed, power usage, and all-around usefulness. The MOSFET, or metal-oxide-semiconductor field-effect transistor, was proposed in 1930 [Lil30], first demonstrated in 1960, and in its current form, started being used regularly in the 1970s. Since the 1980s it has not had a serious rival for building complex computing devices [Key85].

Fig. 4 shows the circuit diagram and two graphs that characterize a typical N-type MOSFET. Each curve in the graph on the left indicates the amount of current that flows from the source to the drain depending on the voltage across the source and drain for a particular voltage applied at the gate. As the gate voltage increases, the amount of current allowed to flow increases, as shown in the graph on the left. The device acts like a voltage-controlled switch. The graph on the right compares the output voltage (V_{out}) as a function of of the input voltage

[2] The vast majority of modern circuits use a clock signal, a signal which arrives at regular intervals, to synchronize the arrival or departure of signals from computation elements.

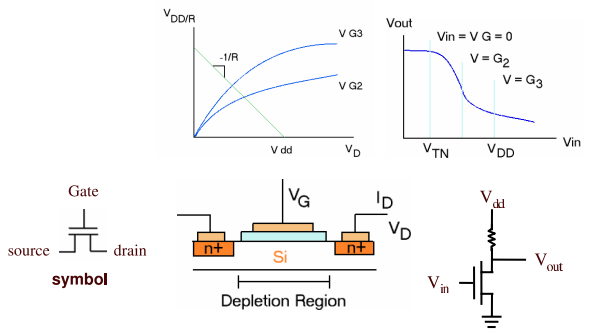


Figure 4: Field effect transistor. The symbol for an *N*-type MOSFET is on the left. A schematic of how it is constructed is in the middle. On the lower right is a circuit used to plot the two curves above.

. As long as the gate voltage is less than a threshold voltage, the transistor is "off." Then, the middle region is reached and the transistor switches "on." Finally, there is a region where the transistor is "on." Depending on the threshold voltage and the shape of the switching region, this device is (with respect to the requirements we enumerated above) an almost perfect building block for logical operations.

An ideal transistor would have its threshold voltage half way between zero volts and the voltage level used to represent a one, V_{dd}. This would give us the most noise immunity. We also want the switching region to be as narrow as possible (i.e., the slope of the curve should be as close to infinite as possible). This slope determines the gain of the device. The gain of a device determines how much amplification the device applies to the input signal. The higher the gain the faster the switching speeds and the more the allowed fan-out.

What is not characterized by these curves is the isolation between input and output. We can see, by looking at how a FET is built (see Fig. 4), that isolation is inherent in the transistor. The gate of an ideal transistor is in effect one plate of a capacitor (i.e., it is not connected to the source or the drain). In practice, particularly as transistors get smaller, there is some "leakage" between the three terminals. This means that as transistors get smaller, they never really turn "off," and there is some connection between the inputs and the outputs.

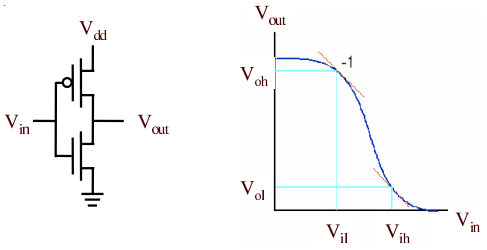


Figure 5: A NOT gate constructed out of CMOS transistors.

MOSFETs come in two basic flavors: *N*-type and *P*-type. *N*-type transistors turn on as the gate voltage increases. *P*-type transistors turn off as the gate voltage increases. The complementary nature of these devices makes it easy to construct restoring logic operations. For example, Fig. 5 shows the transfer curve of a NOT gate built from a *P*-type and an *N*-type transistor. Notice that even when the input voltage is quite far away from an ideal one or zero that the inverter will output a nearly ideal complement of the input. In other words, the output is "restored" back to an ideal one or zero. The output is also clearly isolated from the input.

If we take a closer look at this device we also see that when it has fully switched, it uses no power. For example, if V_{in} is high, then the gates on the two transistors will be charged up. Once they are charged up there is no more current flow to the gates. Furthermore, the *N*-FET is turned on and the *P*-FET is turned off. Thus, there is very low resistance connection between ground and V_{out}. If V_{out} is connected to another gate, then it will discharge that gate through this *N*-fet and then there will be no more current flow. In general, logic built from complementary MOSFETs (CMOS) only uses power when the devices are switching. (As devices are getting smaller and V_{dd} is getting lower this is changing and static power consumption through leakage is becoming a bigger share of the power budget.) What should also be evident from the design of the inverter is that it will switch from high to low as easily as from low to high.

Finally, we can examine how one could create a memory cell. If we connect two inverters together such that the second one feeds back to the first one,

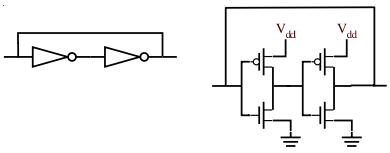


Figure 6: A pair of connected inverters which can store one bit (i.e., a memory cell).

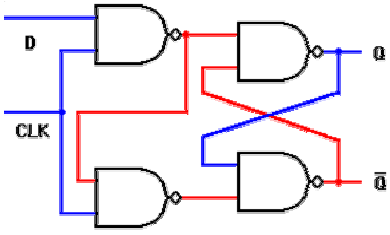


Figure 7: Logic diagram for a D-type latch.

as shown in Fig. 6, we can create a memory cell. If the input to the pair is low, then the first inverter will output a high and the second will output a low, which will feedback to the first inverter. Thus, even if the original input is disconnected the pair will maintain the low output indefinitely, or as long as power is being supplied to V_{dd}. To change the state of the cell, enough current has to be supplied to the input to overcome the output from the second inverter. While these two signals are fighting there will be current flow from the input to ground through the lower right transistor. There are, needless to say, better ways to form storage cells. For example, Fig. 7 shows an example of a D-type latch. These examples bring out an important requirement for building stateful circuits out of logic devices: power gain is required to build a memory cell from logic gates. If the device has no gain, then as the signal circulates through the gates it will degrade, eventually becoming noise.

2.4 Manufacturing and Fabrication

In 1985, Keyes pointed out that the silicon-based transistor, especially as it is manufactured in very-large scale integration (VLSI) chips, is an almost ideal computing device [Key85]. In his article he describes the important attributes of a “good” computer device. In addition to the technical aspects that we have outlined here (e.g., I/O isolation, restoring logic, fan-out, etc.), he goes on to describe how photolithographically produced CMOS also satisfies the manufacturing requirements of a good computer device.

Computer systems currently contain anywhere from thousands to hundreds of millions of transistors. To remain affordable, the individual transistors must be inexpensive to manufacture. For the past 30 years, the cost per transistor has been falling exponentially. Even more important is that the cost of connecting transistors together into entire circuits is also inexpensive and has fallen. The method of making chips, photolithography, is the key ingredient to the ever-decreasing cost of computers. The reason for this decrease is that photolithography combines manufacturing and fabrication.

A raw silicon wafer is converted into a chip with potentially millions of transistors through many steps. Here we describe a gross simplification of the process. After the initial wafer preparation, the structures on the chip are built up layer upon layer using a combination of photomasking and etching. Photomasking involves treating the surface of the chip, then shining ultraviolet (UV) light through a mask onto the treated surface. This cycle is called an exposure. The mask creates a pattern of UV light on the treated surface that alters the surface. Then, some form of etching takes place which affects the light altered surfaces differently than those which were not altered. Three key features of this process are: self-alignment of the transistor gates, in-place manufacture of the components, and integration of transistors and wires. All these processes are similar to photographic development; the most important aspect of this process is that all (up to hundreds of millions of) components are created simultaneously.

There are three contacts that need to be made to a transistor: gate, source, and drain. One goal in constructing a transistor is to keep the overlap between the gate and the source (or drain) as small as possible. The current manufacturing of the MOSFET guarantees this because the gate (formed before the source and drain) is used to create the source and drain. The basic idea is that the outer boundary of the source and drain is defined by the location of the polysilcon that makes up the gate. This ensures that the source and

drain do not overlap with the gate. Equally as important is the fact that the wires that connect to the gate (or source or drain) can also be manufactured at the same time (or potentially at a later exposure) as the transistor is being formed. Finally, notice that all the transistors are made at the same time. In other words, photolithography supports construction of all the devices in parallel. Current chip technologies require more than 20 separate exposures. The result is that the wires and transistors are all fabricated together. This all combines to make each transistor very inexpensive.

In addition to the fact that millions of components are constructed in parallel and that the components are fabricated into a complete circuit, lithography has the advantage that the entire process is very reliable. A chip with millions of transistors and millions of connections will most often have every single one working. Until recently, there has been very little variability between two transistors at different locations in the chip. What little variability was present was overcome by the high gain and use of restoring logic.

The photolithographically manufactured silicon-based transistor has been the dominant building block for digital electronics over the past 30 years. Alternative technologies have either failed to provide the necessary technical specifications (I/O isolation, restoring logic, gain, low-power, high packing density, equal switching times) or the necessary economic advantages (inexpensive to manufacture, reliable, inexpensive to fabricate entire devices). Furthermore, the technology behind silicon-based transistors has been improving all the time, making it increasingly harder for an alternative technology to take hold.

2.5 Moore's Law

The combination of an excellent switch and inexpensive robust manufacturing has led to constant technological advance since the introduction of the integrated circuit. For the past 30 years the minimum feature size that could reliably be created on a chip has been roughly cut in half every 18 months. This has allowed the transistors and wires that make up a computer to shrink in area by a factor of four every 18 months. The feature size reduction is due to continuous advances in manufacturing and quality control. The fact that we can use these smaller devices is due to the abstraction we presented at the start of this section—the bit. This relentless pace of improvement was first observed and predicted by Gordon Moore in 1965 [Moo65]. Since that time, the semiconductor industry's agenda has been set by this "law."

The result of this continuous advance is seen mostly in that the number of components that can be built on a chip grows exponentially with time (see Fig. 8), enabling the construction of more powerful and sophisticated circuits. As the number of components per chip increases, systems which used to require multiple chips now only require a single chip. This collocation dramatically shortens the communication lines between subsystems and causes a boost in performance and a decrease in system cost. For example, until the introduction of the Intel 80386 microprocessor, the floating-point operations of Intel processors were carried out on a separate slave chip, called a floating-point coprocessor. The next generation integrated the two on a single die, achieving a performance boost. Power consumption per performance also decreases with feature reduction, because smaller circuits can use lower supply voltages and thus dissipate less power when carrying the same computational task.

Surprisingly, despite the exponential feature reduction, the actual total chip size and total dissipated power of microprocessors have kept increasing with time. The main reason is the integration of more and more of the functionality on the same die. Another reason is that microarchitects have used the available real estate to implement more computational and support structures for the execution of multiple operations simultaneously.

All of these effects arise because as chipmakers decrease feature size they also maintain yield. Yield is defined as the percentage of manufactured chips which are defect-free. The importance of defect control in integrated circuit manufacturing cannot be overemphasized: higher defect rates translate into higher unit costs. The cost of an integrated circuit

is proportional with the manufacturing cost, but inversely proportional to the yield. As we discuss more fully later, the nature of digital integrated circuits makes them very brittle: even one minute defect can make the whole circuit unusable. In consequence, a high yield is extremely important for keeping costs low. Because of the microscopic size of the electronic devices, a speck of dust or a tiny misalignment in the photolithographic masks can create irrecoverable defects. These requirements drive up steeply the cost of production plants: modern clean room facilities, using extremely precise mechanical and optical equipment, cost several billion dollars.

Despite the microscopic size of the basic components, extreme care in manufacturing has managed to maintain the yield practically constant in the last decade. However, two factors make it unlikely that this trend will continue for too long: (1) as the number of components on the same area grows exponentially, the probability that none is flawed decreases; and (2) as the size of the individual components decreases, they become increasingly sensitive to impurities (e.g., minute dust particles, radiation, etc.).

2.6 The Future

Impressive as these results have been, in the near future further increases in the performance of silicon-based, lithographically manufactured transistors will be difficult to achieve. Never before has there been so much doubt in the industry about how advances three-generations in the future will be accomplished [Sem97, Semte]. There are several major reasons for this. First, the small linewidths necessary for next-generation lithography require the use of increasingly shorter wavelength light, which introduces a host of problems that are currently being addressed [Pee00]. In addition, as the number of atoms that constitutes a device decreases, manufacturing variability of even a few atom widths can become significant and lead to defects.

More important, however, is the economic barrier to commercial nanometer-scale lithography. New fabrication facilities orders of magnitude more expensive than present ones will be needed to produce chips with the required densities while maintaining acceptably low defect rates. The increasing cost of chip masks, which must be manufactured to single-atom tolerances, precludes commercially viable development of new chips except for the highest volume integrated circuits. It is entirely possible that further reduction of transistor size will be halted by economic rather than technological factors.

This economic downside is a direct consequence of the precision required for deep-submicron lithographic fabrication. Using lithography, construction of devices and their assembly into circuits occur at the same time. Keyes pointed out that this is a great advantage of silicon because mass fabrication of transistors is extremely inexpensive per transistor [Key85]. However, as Keyes also highlighted, it produces a set of constraints: each element in the system must be highly reliable, and individual devices on a chip cannot be tested, adjusted, or repaired after manufacture. These constraints force the design of a custom circuit to be tightly integrated with manufacture, since no additional complexity can be introduced into the circuit afterward. Lithography is perfectly tuned to deal with these constraints, but at small scales the precision required to obey these constraints becomes the obstacle to further improvement. Any technology hoping to displace lithographically produced CMOS integrated circuits must overcome such obstacles without sacrificing performance or the low per-unit cost made possible by mass production.

2.7 A New Regime

As we have already suggested, we are entering a new regime—a regime where the transistor, and particularly the photolithographically produced transistor, may not be king. Since all the attributes of a good computing device have not changed, we need to examine the assumptions on which the transistor's dominance is based. For example, one of the key assumptions made by Keyes in 1985 is that the individual devices cannot be tested nor altered after fabrication. This assumption is what leads to the need for high noise margins, high gain, reliable construction, and low manufacturing variability.

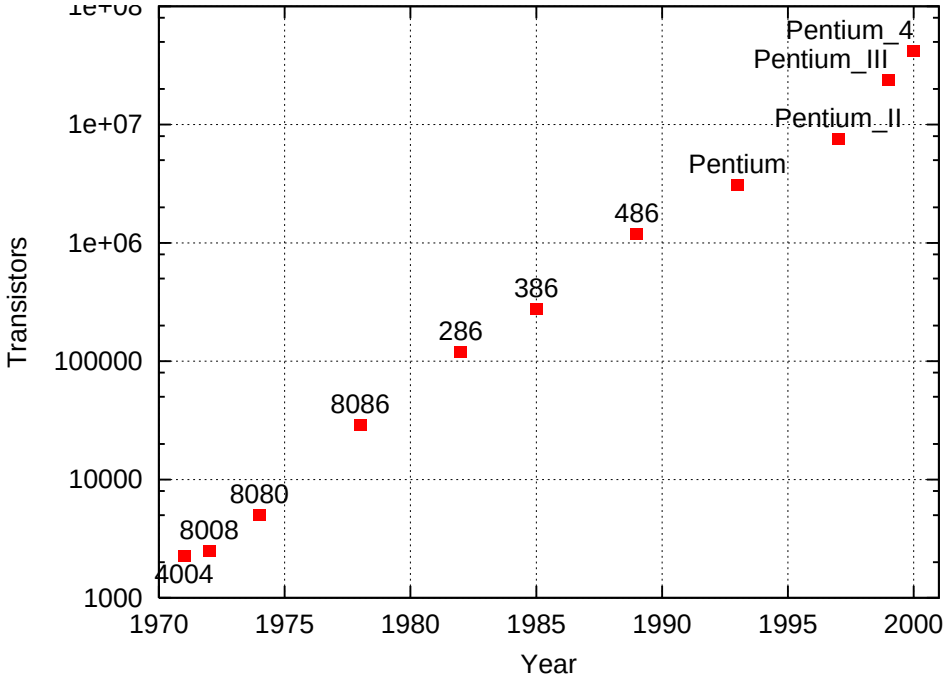


Figure 8: Moore's law describes the exponential increase of the number of transistors per unit area. This graph depicts the size of successive microprocessor generations of Intel's x86 family.

Current chip manufacturing is based on the idea that the functionality of the chip is fixed at manufacturing time. In other words, the chip is designed, masks are made, and the chip is fabricated, tested, and then put in a package. If there is an error at any stage in the process, the final device is inoperable. This means that every transistor and wire in the fabricated chip needs to be highly reliable as a single point of failure can cause the entire chip to be defective. An alternative system would be to draw on the idea of general purpose computers, allowing a chip to have its functionality altered after it is fabricated. In other words, we envision a chip as a piece of reconfigurable hardware (see Section 5).

Reconfigurable hardware reduces the need for reliable components because if a particular component on the hardware is faulty, we can simply avoid using it when the hardware's final functionality is determined. In other words, we configure around the faulty component, allowing chips with many defective parts to perform their desired functionality. In Section 5 we also show how such hardware can test itself, eliminating the requirement that we are not allowed to probe the components.

Relating to the information theory description at the start, we can talk about when information is added to the system to create a working chip. The current method adds all the information at or before manufacturing time. Our method allows information to be added after the time of manufacture. This reduces the cost of manufacture substantially. It also increases flexibility and reduces design and testing costs.

If we are going to replace photolithography and the CMOS-based transistor, we need to keep in mind the qualities that any computer requires in it switches. Ideally we hope that they:

- support restoring logic
- isolate the inputs from the outputs
- provide gain
- allow a complete logic family to be constructed
- are inexpensive to manufacture
- use little power (particularly when not switching)

- pack densely
- are reliable

Furthermore, it should be inexpensive and reliable to connect them up into a complete circuit.

Let us examine these "requirements" in turn. Restoring logic is necessary in order to support abstraction and the design of complex, yet, reliable systems. Similarly, input/ouput isolation is necessary to support the design of large systems. This requirement could, however, be too stringent. For example, imagine a system in which some elements are restoring and others are not. If the elements that restore logic levels are used frequently then it would be possible for some of the switches to not restore logic levels, similarly for isolating the inputs and outputs. In other words, the granularity of the restoring logic component could be raised from one to a few without sacrificing the ability to abstract. Similarly, as long as there is some element that can isolate elements of the circuit from each other the designer can safely ignore the effect of the rest of the circuit on a localized structure.

Gain is required to build memory devices and to tolerate noise and manufacturing variability in the environment. If we can construct a system which has a memory element as an atomic element, then gain becomes less crucial (i.e., other fault-tolerance mechanisms can be used to overcome a lack of significant gain). We discuss fault tolerance in Section 4.

Earlier we described a system of complete Boolean logic. For example, an AND and a NOT gate form a complete system. Actually, this assumption is also too stringent if we require that all inputs to the system appear in both their true and complemented form. That is, instead of computing just $f(a, b)$ we compute functions $f(a, b, \overline{a}, \overline{b})$ and $f(a, b, \overline{a}, \overline{b})$, then a complete system is one which includes only an AND and an OR gate.[3] In other words, if we consider a system which includes inverters on the inputs to the system, the internal parts of the system do not need to have any inverters.

[3]This can be shown by using de Morgan's law: aANDb equals $\overline{\overline{a}}$OR$\overline{\overline{b}}$. $\overline{a}$ should be read as "Not a."

From this analysis it appears that what we absolutely need are devices that pack densely, are low power, and are inexpensive to manufacture. Also, the fabrication of entire devices should be inexpensive as well. Luckily, these requirements are met by chemically assembled electronic nanotechnology.

3 Processor Architecture

In this section we briefly present basic concepts of computer architecture, with an emphasis on processor architecture. We discuss the evolution of processors and identify some hard problems faced by future designers of computer systems. CAEN has the potential to efficiently solve some of these problems.

3.1 Computer Architecture

3.1.1 Computers as Universal Machines

Computers are *universal machines* because they can be "taught" (i.e., programmed) to implement any task we can describe algorithmically. Computers were not always this way: the first computers were just collections of functional units which were wired together manually to implement the desired computations. For example, if you wanted to subtract two numbers and then square the result, you would hook the wires fetching the numbers into a box which subtracts them, and you would wire the output of the subtracter to both inputs of a box doing multiplications, as in Fig. 9a. If you wanted a different computation, you had to unhook the cables connecting the units and plug them in a different way.

Eventually a great insight occurred: a computer can be made to carry out different computations each time without any modifications to the underlying hardware. This finding is attributed to John von Neumann, although the idea existed in abstract terms long before. Von Neumann realized that the way a computer should behave can be described by a *program*, which can be stored in the memory of the computer [vN45]. By changing the program, you can change what the computer was doing.

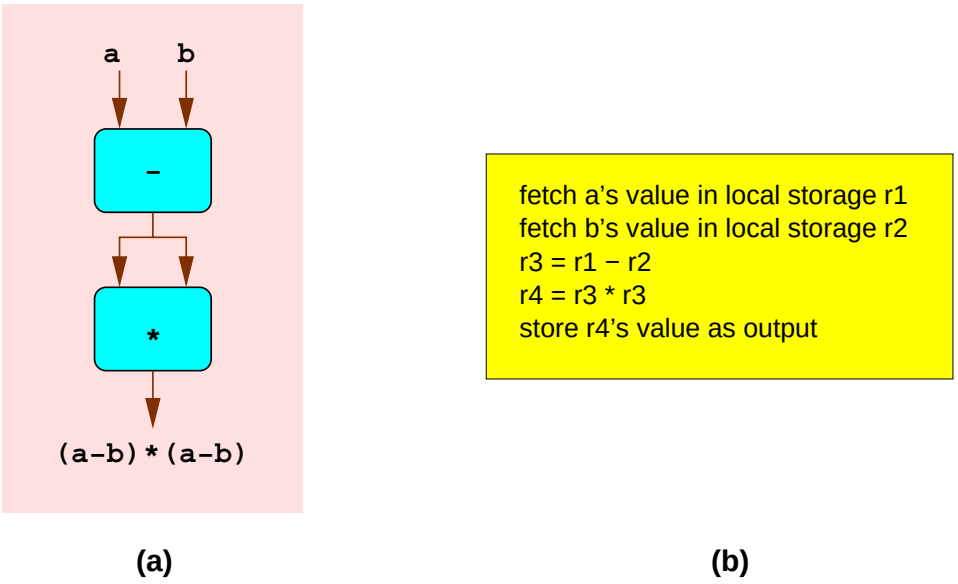


Figure 9: (a) Hardware implementation of a circuit computing $(a-b)^2$. (b) Software program performing the same computation.

Programs are sequences of simple instructions which direct the way data is routed through the computational units of the machine. For example, the previous computation would be implemented by a program having instructions to fetch the two data values from storage, feed them to a subtracting arithmetic unit, and feed the result to a multiplier, as in Fig. 9b.

Most modern computers are based on von Neumann's idea. Their instructions are stored in a memory, which is also used to store data and intermediate computation results. Computers thus consist of: memories, storing data and programs, peripheral units, interfacing the computer to the outside world, and a central processing unit, the computer's brain, which manipulates the data according to the program instructions (see Fig. 10). In modern computers the central processing unit consists of one (or sometimes multiple) microprocessors.

Von Neumann architectures have the tremendous advantage of flexibility, as the functionality of a computer can be changed just by loading a different program, without any hardware modifications. However, this flexibility results in a weakness, called the "von Neumann bottleneck" [Bac78]: there is inherently a sequential flow of information between mem-

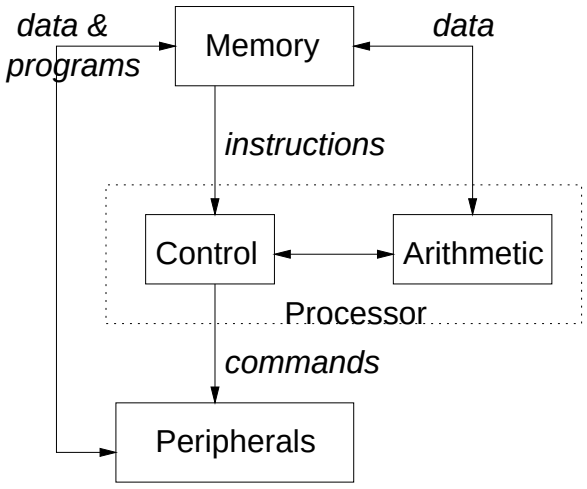


Figure 10: In von Neumann architectures, programs and data are objects of the same kind and are stored in memory. The program is composed of instructions which indicate how data is to be processed.

ory and processor. Instructions and data are brought essentially one-by-one from memory and processed by the processor. Computer architects have enlarged somewhat the width of the communication channel between memory and processor. But the bottleneck exists in today's computers and has become one of the major limiting factors of computer system performance.

3.2 Instruction Set Architectures

In 1964, the IBM System/360 introduced the idea of a family of computers that would all execute the same instructions. The instruction set constitutes an interface between the microprocessor hardware and software. This interface is called the instruction set architecture (ISA). It is a *contract* between the two worlds. This interface is similar to the electrical plug in our homes: the electricity can be produced in various ways (using coal, atomic energy, or renewable resources) and can be consumed in countless other ways. However, the interface delivering the electricity is precisely standardized, enabling an independence between producer and consumer.

This independence is also true for the hardware and software worlds: for example, all Intel processors, starting from the Intel 8086 built in 1978, until today, have implemented practically the same ISA called the x86 ISA.[4]

The ISA standardization is very important economically, because it decouples the software and hardware producers: as long as the ISA is precisely specified, different manufacturers can compete to produce hardware implementing it and software using it. For example, today Intel, Advanced Micro Devices, and Cyrix all produce implementations of the x86 ISA, while countless software producers create programs using it. The existence of an ISA makes hardware and software development independent processes.

[4] In truth, the ISA has changed slightly from processor to processor, but in a backward-compatible fashion, by always adding new features. This is much like how the method of adding a third prong to a wall outlet extends the functionality of the outlet, but continues to allow you to use two-pronged devices.

In this section we describe briefly the components and implementation of a modern ISA.

3.2.1 Instructions

Modern ISAs differ from one another more in details than in general structure. We can distinguish several types of instructions: arithmetic and logic, memory access, control flow, and input-output.

Basic arithmetic and logical operations do not require further elaboration. Arithmetic is carried out on finite value integers, usually 32 or 64 bits long.

Another important type of operation is *memory access*. The memory of the computer is viewed like a large array of data words, each having a numeric *address*. Memory access operations allow the processor to compute memory addresses and retrieve the contents of the specified memory cells.

A special class of instructions, called *control instructions*, indicates to the processor which instruction to execute next. When executing these instructions, the processor can branch to one or another instruction, depending on the outcome of a previous computation.

Basic arithmetic operations, control instructions, and memory access are the only required ingredients for building a truly universal computational machine. Any computational task can be described as a sequence of these operations.

However, the computer must also carry out non-computational tasks, such as "talking" to the peripheral devices. The peripherals are accessed through a special class of input/output instructions, similar to memory access instructions. However, these instructions write and read data from external devices rather than to memory.

For reasons of efficiency, some common complex tasks have been encoded in single instructions; for example, modern ISAs also deal with:

- floating point computations, which are performed on finite-precision representations of real numbers

- special “media extension” instructions, used by algorithms manipulating digitally encoded media such as speech or images
- instructions aiding the implementation of the operating system, which offer help in creating the virtual machines protecting programs from one another
- exceptions and interrupts, which are events through which programs or peripheral devices signal to the CPU that something unpredicted has happened, requiring immediate attention and the interruption of the currently running computation

Instructions are represented in the computer memory by binary numbers and programs are sequences of instructions. To help humans create and manipulate programs, a set of software tools helps translate programs written in textual form to and from these binary representations. One form of textual program representation is “assembly language.” Assembly language instructions stand in one-to-one correspondence with the ISA operations: each machine instruction is represented by a corresponding textual description.

To illustrate, the program in Fig. 11 computes the sum of the numbers between 1 and 10 is written in the x86 assembly language. At the end of each line is a comment, human-readable description of the instruction and its indented action. This program features arithmetic and control operations. Data values are stored and manipulated in internal microprocessor registers; in this code fragment we see two registers called `eax` and `edx`. We also see a branch instruction, which uses the result of a previous comparison to decide whether to re-execute a code fragment. As long as register `edx` has a value less or equal to 10, the `jle` instruction steers execution at the `add` instruction. This will happen exactly 9 times—causing the `add` to be executed 10 times.

3.2.2 Microprocessor Operation

A microprocessor starts executing a program when the computer is started and stops only when the computer is shut down. The processor continuously repeats the same tasks [HP96], depicted in Fig. 12, which can be described as follows:

Fetch: Obtains the next instruction to execute from memory.

Decode: Determines the instructions desired functionality: which data is to be operated upon, what operation is to be carried out, where the results should be sent. Decoding generates signals which steer the data through the processor.

Read: Retrieves the data to be operated upon by the instruction, either from memory or from registers.

Execute: Performs the main action of the processor, which consists of generating a result value based on the combination of operands and the selected functionality. In this step “new” information is generated.

Writeback: Commits the result of the instruction. Once the result is known, it is sent to the destination indicated by the instruction: either a memory location or a register. The result overwrites the previous contents of the storage location.

3.3 Processor Evolution

In this section we briefly overview the evolution of microarchitecture since 1971, when Intel introduced the first microprocessor. It is impossible to do justice to such a broad subject in this short space, so we simply identify the major paradigms used in building processors today.

The evolution of computer architecture has been driven by the insatiable need for more computing power. The truth is that we will never have enough computational power, because important classes of practical problems are very hard in a computational sense [GJ79]; that is, they require a very large amount of computation, which grows rapidly as we increase the amount of manipulated data (e.g., exponentially or even faster).

```
        mov 0,eax       # register eax is initialized with zero
        mov 1,edx       # loads the value 1 in register edx
L34:                    # label used for branching
        add edx,eax     # add contents of register edx to register eax
        inc edx         # increment (add 1) contents of register edx
        cmp edx,10      # compare contents of register edx with 10
        jle L34         # if result is less or equal jump to label L34
```

Figure 11: An assembly language program to calculate the sum of the first ten integers.

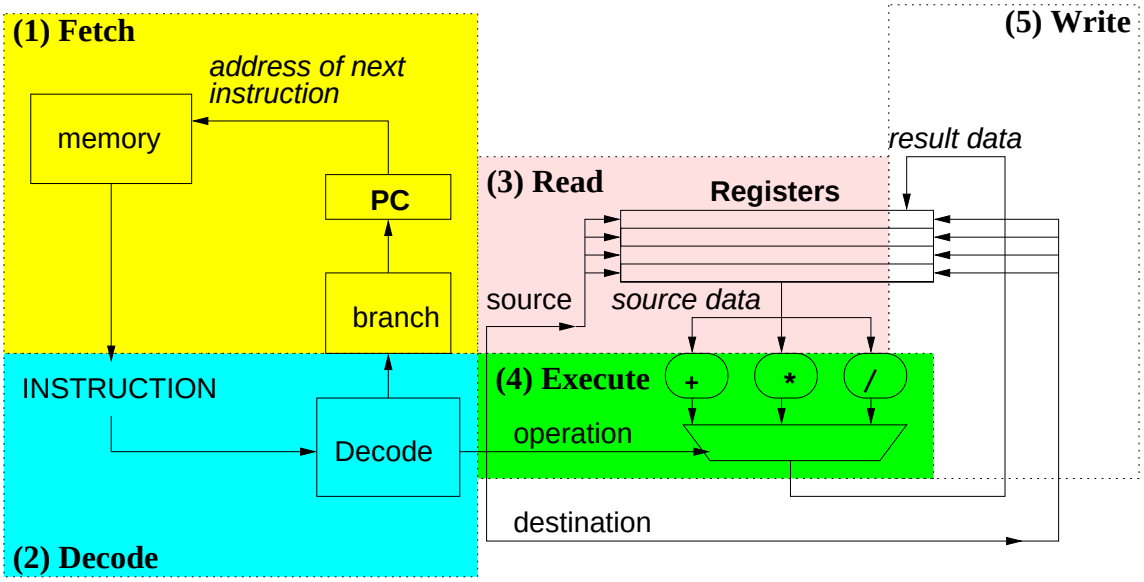


Figure 12: The phases of execution of an instruction: fetching instruction from memory, decoding, reading the operand data, executing the indicated operation, and writing the result.

Our discussion of microarchitecture evolution will therefore be centered on its impact on performance. Most of the microarchitectural innovations were actually introduced for increasing the system performance. The performance of a processor is measured in the number of instructions it can execute in a time unit. It is thus a measure of the *instruction throughput* of the processor. The unit of measure is million instructions per second (MIPS) [HP96]. MIPS is an imprecise unit because instruction throughput depends on the executed instruction mix (i.e., a processor may be capable of performing many more additions than multiplications in a given time interval). Nevertheless, it is still informative.

3.3.1 Clock Speed Increase

One of the most publicized marketing wars of the last few years has been the "war of the megahertz." The clock of a processor is used to synchronize the computation of all the components on the die; it is therefore a coarse measure of performance.[5] Indeed, the performance increase derived from speeding the clock signals is truly impressive: in 31 years since the introduction of the first microprocessor, the clock speed went from 740 kilohertz to 3 gigahertz, which is an increase of 4000 times!

Clock speed increase is driven by miniaturization: as the components shrink, the distances traveled by the electrical signals in a clock cycle also decrease, allowing the use of clocks with shorter periods. Significantly, however, clock speed has already reached some fundamental limits. Given the propagation speed of the electromagnetic signal through the logic gates and wires, at current clock speeds the signal can barely cross from one side of the chip to the other. If clock speed increase continues to keep the same pace, in 10 years a clock cycle will permit the electrical signal to reach less than 1% of the whole chip surface [HMH01].

3.3.2 Pipelining

Pipelining is the name used in computer architecture for the assembly-line method of manufacturing. When building cars on an assembly line, car frames are moved from worker to worker. Each worker is specialized in only one operation: adding doors, checking the engine, etc. More importantly, a car can have its doors mounted while another one is having the engine checked. The two activities can thus be performed at the same time. While the time to make a single car is unchanged, the time between the completion one car and the next is significantly shortened.

As mentioned in Section 3.2.2, the "work" to be carried out on each instruction usually consists of five phases: fetching, decoding, reading, executing, and writing.[6] If we associate the hardware doing each operation with a worker and the instructions with cars, an implementation of the assembly line gives us a pipelined processor (see Fig. 13).

Pipelining is a straightforward concept, providing great benefits with relatively little effort. It was first investigated in the 1960s and has been universally used since the beginning of the 1980s. In pipelining, instructions enter and exit the pipeline in the order they should be executed. However, a "clumsy worker" at the beginning of the pipeline can reduce the performance of the whole system because all later stages must wait for it. Unfortunately, while executing programs the instruction flow may be interrupted for a number of reasons. For example, when a control instruction (branch) is executed, the proper following instruction to execute will not be known until the branch completes. When a branch enters the pipe normally no other useful work can be performed until its result is known.

To address the events which prevent the pipeline from working at full capacity, microarchitects resort to very complicated techniques, some of which are described below.

[5] There is no intrinsic number of cycles required for a computation: for example, on some processors a 32-bit multiplication is implemented in 4 clock cycles, while on others, it takes 16.

[6] For some instructions, some phases may be missing, while for other instructions, some of the phases may be further decomposed in subphases.

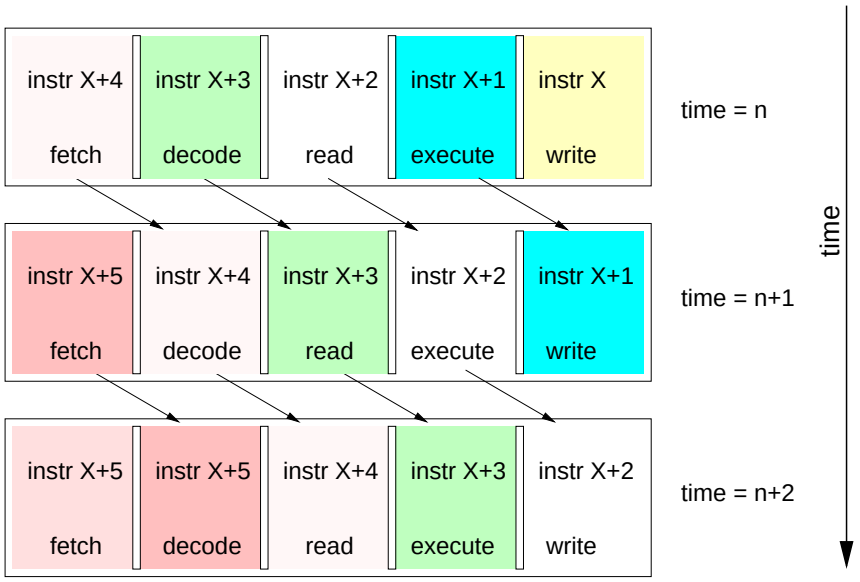


Figure 13: Pipelined microprocessors have multiple instructions in flight, each in a different execution stage.

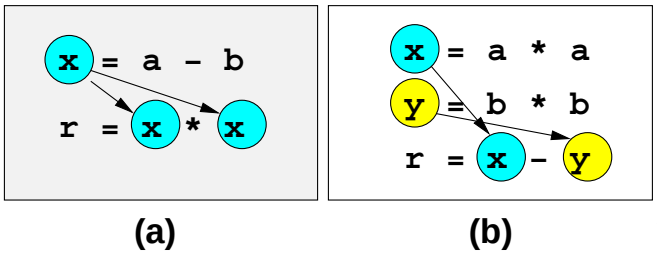


Figure 14: (a) Two consecutive dependent instructions. (b) The first two instructions are independent.

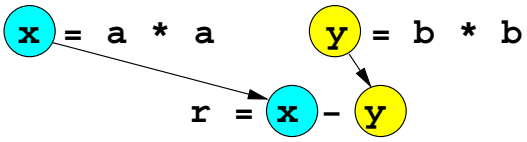


Figure 15: (a) Some of the instructions in Fig. 14b can be executed in parallel because they are independent.

3.3.3 Parallelism

It is generally accepted that computer programs contain many operations which are not interdependent.

To illustrate, let us consider two examples. If we want to compute $(a-b)^2$ (for some given values for a and b), we need to make a subtraction and a squaring, and the squaring cannot begin until we know the result of the subtraction. We say that these operations are *dependent*. However, when we want to compute $a^2 - b^2$, we can give each squaring to a different person to compute at the same time (as in Fig. 14).

Exploiting independent instructions by executing them simultaneously (as in Fig. 15) is the most important trend in microprocessor architecture in the last 20 years. This type of execution exploits *instruction level parallelism* (ILP) [RF93].

There are other types of parallelism which are exploited in computer systems to increase parallelism: for instance, data width increase is a form of bit-level parallelism, where computations occur on more bits in a word at once. Another type of parallelism, exploited in a system having multiple microprocessors, is process-level parallelism[7] because multiple programs are executed simultaneously, one using each processor.

In this section we briefly overview several techniques used for the exploitation of parallelism.

Word Width Increase Microprocessors can be characterized by the basic word size they operate on,

[7] A running program is called a "process."

measured in bits. Intel 4004 is a 4-bit microprocessor because an addition computes the sum of two 4-bit numbers in one operation.

A word has two important uses: it is the basic unit of data on which the processor operates and it is also the basic value used to specify a memory address (when using indirect addressing). As a consequence, the size of the memory attached to a computer is limited by the word size. A processor with a word of 16 bits cannot express binary numbers larger than 2^{16}, which puts a limit of 64 kilowords on the addressable memory.[8] The latest processors supporting the x86 ISA are 32-bit processors. The quick increase in memory capacity makes 32 bits too limiting, capping the addressable memory to 4 gigabytes. Workstation microprocessors already have 64-bit words and there are proposals (most notably by Intel's rival, AMD) for extending the x86 ISA to 64 bits as well.

It has been empirically noted that miniaturization permits the growth of memory sizes with about 1 1/2 address bits every two years, keeping cost constant. At this rate, the jump from 32 to 64 bits should buy computer architects about 48 more years before 64-bit address spaces become insufficient. Widening the word size implicitly grows the computational power of the machine: adding two 64-bit numbers requires two 32-bit operations but is workable using a single operation on a 64-bit processor.

Despite the fact that memory capacities have grown at an exponential pace, the size of the problems we are solving and the nature of current applications have always immediately taken advantage of the increased capacities. For example, today's computers manipulate not just text and graphics, but also sound and video, which are huge storage consumers.

Very Large Instruction Word To return to the auto manufacturing example, another natural way to speed up car production is to assign several workers at once to each car. Each worker is given an independent task, they all work in parallel, and before anyone starts on a new task, they wait for all the other workers to complete their task. As long as the workers are independent, and the tasks assigned are all fairly equal in the time they take, then the presence of more available workers results in the production of more cars per hour. One can see that for the factory to run smoothly the scheduling of the tasks to workers is essential.

To apply this idea to processors, we just need to have multiple processing units (workers) available. However, because not all instructions can be processed in parallel due to the dependencies, we must first group independent instructions together. When this grouping must be done by the compiler—because the multiple units all work in lock step—the processor is called a very long instruction word (VLIW) machine. The terminology indicates that the machine really executes superinstructions, which comprise several normal independent instructions. VLIW machines are widely used in signal processing tasks, for example, most cellular telephones have such microprocessors. Compilers for VLIW not only group together independent instructions but also choose the groupings based on the expected execution time of each instruction, planning ahead to best utilize the processing units. The compiler operation which selects the instruction order is called *scheduling*. Just as in the factory above, the scheduling done by the compiler is an essential of an effective VLIW processor. In practice, VLIW machines are also pipelined, having multiple parallel pipelines.

Superscalar Processors VLIW processors work very well as long as the compiler can predict how long each job will take. Then the available workers can be assigned very efficiently to the available tasks. However, real life is always more complex and in a processor unpredictable events may occur. Further, the duration of each instruction may not be easy to predict. In such a case the carefully constructed VLIW schedules become inefficient.

To overcome the scheduling difficulties introduced by unpredictable latencies, computer architects have proposed superscalar processors: these processors dynamically compute dependencies and schedule instructions as soon as all their operands are available.

[8] Very old processors used tricks to express addresses as combinations of multiple words to overcome the word size limitation.

Much of the work of the compiler scheduler is moved in hardware in these machines.

Like VLIW machines, superscalar microprocessors have multiple processing units (workers). Unlike VLIW machines, each idle worker can begin a task as soon as there is something available to perform. The length of time each worker takes to process each instruction, or which worker will deal with which operation, does not need to be and in general cannot be predicted in advance. Most desktop and workstation processors today are superscalar.

To implement dynamic instruction scheduling in hardware, the processors use a series of very complex hardware structures. These structures keep track of the status of each instruction. The status includes which ones have completed execution, which are waiting for data, which are currently in execution, and which ones have not started yet. The instructions waiting for inputs monitor the instructions under execution. When these complete and write their results, instructions waiting for these results fetch them and update their state correspondingly. If all of their inputs have become available, they are marked as ready for execution.

Prediction and Speculation No matter how many computational resources we use, if there are many dependent instructions in a program, there is not much that can be done: we must wait for an instruction to complete its computation to give the result to the dependents. Or do we?

Actually, we can do even better! Instead of keeping the functional units idle, we can simply guess what the result of a long latency instruction is and start executing its dependents immediately, using the guessed value as input. When the instruction actually terminates, we check whether we correctly guessed its output. If the guess was correct, we have saved time because we have already started the dependents. If the guess is incorrect, we must unfortunately discard the work of the dependents. We start them again, using the correct value of the output this time. This strategy has been shown to be beneficial because correct guesses occur frequently enough to achieve important speed-ups.

Microarchitects have incorporated this technique in processors in multiple guises—guessing is officially called *predicting* and executing ahead of time is called *speculating*. The most frequently used predictors are used to guess which way branch instructions will go [MH86] to be able to start execution from the target instruction early.

One way of using speculation in order to remove control dependencies is depicted in Fig. 16. In this example there is no prediction because both possible branches of a conditional are executed.

While we noted earlier that there is not much to lose by guessing, speculation does have a downside: it requires very complicated hardware resources to track the predictions and to flush out the speculative results when predictions are wrong.

Speculation also interferes with other tasks of the processors: for example, instructions executed speculatively cannot simply complete until it is known whether they should have been executed in the first place. This raises many issues if these instructions trigger exceptional events (such as a division by zero): the treatment of these events must be postponed until the prediction has been validated.

Thread-Level Parallelism Instruction-level parallelism is not the only type of parallelism that can be exploited to speedup execution. What if we could decompose an application into multiple relatively independent activities which can be carried out in parallel? A typical example is a web server: it has to do some work for each request coming from a web client. The entire work for several clients can be done completely in parallel because the clients' actions largely do not interfere with each other. This type of parallelism is called *thread-level* parallelism: one way we can view a threaded program is as being composed of a multitude of identical programs, all executing the same code, but manipulating different values.

The newest processors have architectural support for multithreading. Multithreading is used to cover for the periods of inactivity caused by long-latency instructions: when a thread has to stall, for example because a long-latency instruction did not retrieve its

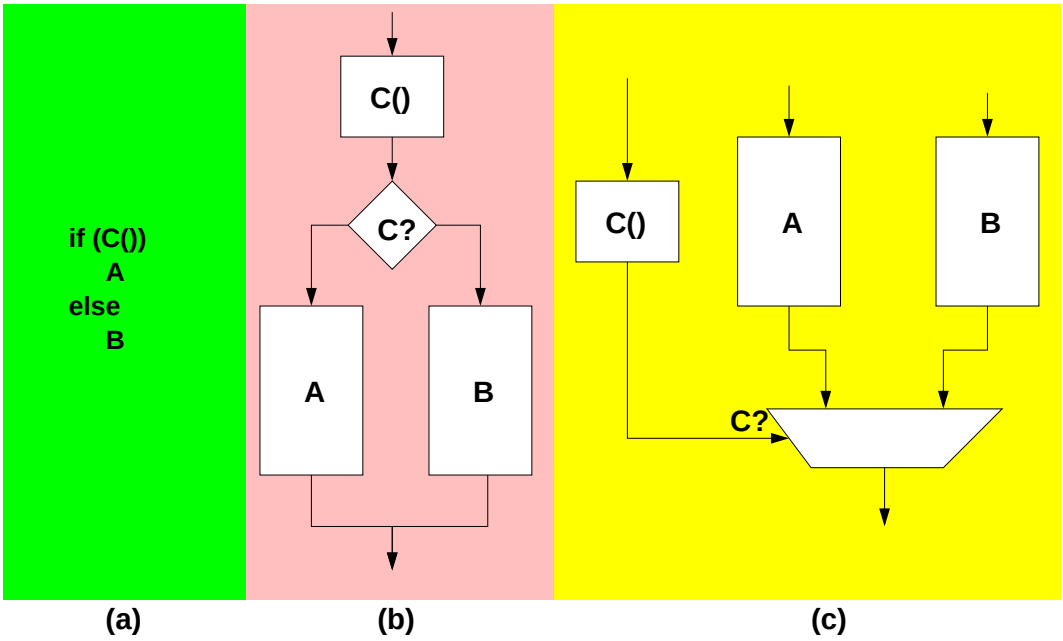


Figure 16: (a) Program fragment, (b) normal execution, (c) speculative execution. In the speculative version, both targets of a conditional branch and the condition itself are executed simultaneously. This removes the requirement for serializing the execution of the branch and of the target code.

data fast enough, another thread can use the idle processor to run its own computation [TEL95].

The problem of multithreading is compilation. Although we can write compilers to extract ILP from programs, the quest for a compiler which decomposes a single application into multiple threads automatically is much farther from fruition. Moreover, manually writing programs with multiple threads is a very cumbersome task; it is difficult for humans to think about multiple simultaneous activities that might interfere with eachout.

Multiprocessors When computers have several different jobs to perform, they can benefit from the existence of multiple microprocessors. While multiprocessor systems are presently a high-end feature, with the pace of feature shrinkage we can expect in the near future that such systems will be common in ordinary desktop computers.

3.3.4 Caches

In the beginning of this section we mentioned the von Neumann bottleneck: the fact that the processor has to bring instructions and data from memory and to store computation results in memory. We further noted that memory access must inherently be done sequentially. To compound this bottleneck, the relative speed of memory versus processor has been decreasing at an exponential pace [HP96]. However, processor and memory access speed have both grown exponentially with time. While the processors boast a respectable 55% increase per year, the time taken by a memory chip to service a request shrinks only about 10% each year. This means that relative to the processor, the memory is slower and slower.

The memory commonly used in a computer system is of a type called Dynamic Random Access Memory (DRAM). DRAM is relatively cheap and compact, but is slow (and relatively slower with each new generation). Computer architects have another type of memory at their disposal, Static RAM (SRAM). SRAM is bulky and much more expen-

sive than DRAM, although its performance increase keeps pace with the speed increase of the processor.

The first microprocessor generations were slow enough to use just plain DRAM: their clock cycle would allow memory accesses to complete during the time of one processor operation. However, the speed increase disparity soon led to a crossover point, which happened at the beginning of the 1980s. Since then processors are faster than memory. Today, a memory access can take as many as 500 processor cycles. This means that the processor wastes 499 cycles for every access to DRAM. Computer architects came up with a compromise solution which trades cost for speed by creating an intermediate static RAM layer between the processor and the main memory. This intermediate layer is called a *cache*.

The cache is used much like students use their short-term memory before an exam when they cram and memorize the information required for the exam. The information is then discarded the following day, when they start cramming for the next test.

Whenever the processor needs data from memory, the data is automatically brought into the cache by some helper circuitry and served to the processor from there. This is not much benefit unless the processor asks again for the same data item soon: the second time it will be brought directly from the cache, much faster than from memory (see Fig. 17a). This goes on until the cache gets full (which eventually happens since it is much smaller than the main memory). Afterwards, when even more data is accessed, something must be evicted from the cache to bring in the new data.

Caches are beneficial when data access exhibits a lot of "reuse," or *locality* in computer terminology. Empirically it has been noted that most programs indeed have a high degree of locality and therefore benefit highly from caches. However, some applications (of which media processing is an important class) exhibit little reuse, so they cannot really benefit from caches. For example, viewing a movie on the computer displays each frame and then never uses that data again.

The importance of caches has grown steadily with time. As memory size grows, and the speed disparity widens, the trend is to add yet another layer of cache between processor and memory, the "outer" layers (i.e., closer to memory, in Fig. 17b) being larger, cheaper, and slower. When miniaturization progresses enough, system integration moves the caches on the same integrated circuit with the processor and inserts new cache layers outside. Modern workstation processors have three layers of cache: the first two reside on the processor die. Access times to main memory can be several hundred processor cycles, so performance in the absence of caches would be abysmal.

Creating more and larger caches is not the only evolutionary trend: new caches are also substantially more sophisticated than the original ones. For example, modern caches can continue to serve requests for data even after some data has not been found (because it did not reside in the cache). The cache then simultaneously communicates with the processor and to the memory, maintaining queues of pending requests from either side and handling all of them in any order.

A complication arises when we build computers having multiple microprocessors, as is increasingly customary. Then each processor will have its own caches, which can lead to confusion when one processor intends to change data: the changes should not be confined to the copy of the data in the local cache, but propagated to the remote processors as well, and all this should be done without incurring too much overhead. This is called *cache coherence* and is the main reason for which computer systems with many processors are very hard to build. The commonly used cache coherence protocols simply cannot accommodate too many participants without incurring prohibitive performance penalties.

Caching is a technique used very frequently in computer system design, at all levels. For example, there are caches between the main memory and the even larger, slower, and cheaper disks. With the emergence of the Internet, caches are also used to maintain local copies of data brought from the network. There are many other examples of caching in use today.

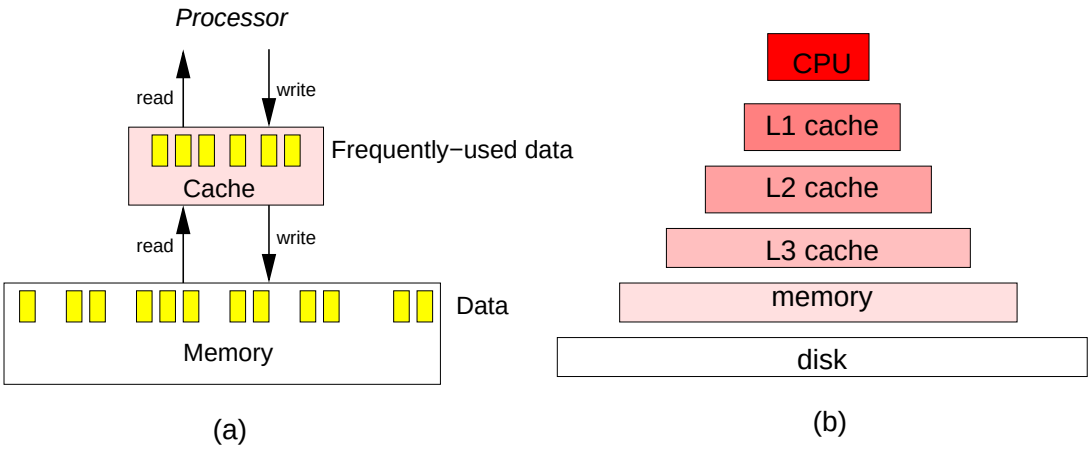


Figure 17: (a) Caches offer the same operations as memory: reading and writing of data. However, they are faster and smaller in capacity for two reasons. Their small size allows electrical signals to traverse them faster. Also, they are smaller because they are more expensive per storage unit. (b) Modern systems feature multiple layers of cache; the ones closer to the processor are faster and smaller.

Conclusion Modern superscalar processors use all of the above techniques simultaneously to increase performance: they contain multiported register files (which allows multiple instructions to access registers at once), level 1 and level 2 caches for instruction and data, issue logic (which dynamically computes dependencies between instructions), forwarding logic (used for bypassing dependencies between instructions simultaneously in the pipeline), branch prediction logic (which monitors the execution of the program and tries to guess the direction of the following branches, to pre-fetch the destination instructions), and many others that consume most of the area and energy, while only supporting the computation.

For example, the newer processors can execute multiple operations in parallel, while the old ones had only one arithmetic unit, so they could execute at most one instruction at a time. Most resources in the new processor are devoted to the cache. The complete removal of the cache would not change the functionality of the processor in any way but it would adversely impact its execution speed.

Describing the way the processor computes was a relatively straightforward task for the 4004. However, it is an extremely complicated process for the Pentium 4. While the 4004 would basically carry the five-step process described in Section 3.2.2, the Pentium 4 simultaneously reads multiple instructions, checks if they depend on each other, tries to obtain the operands of all of them (some of them may need to be obtained from the cache, some may be in memory, some may be in registers, while others may be in the middle of the pipeline), dispatches them to multiple functional units whenever their inputs have arrived, checks the completion of each instruction, and launches other instructions even if the previous ones have not yet completed. In all, Pentium 4 can have more than 100 instructions “in flight” at any one time.

3.4 Problems Faced by Computer Architecture

Despite the amazing progress of computer system performance, the horizon is darker for computer architects. The “easy” methods for performance enhancement have been exploited almost to their limits and we are now left facing very hard problems. In this section we discuss briefly some of these problems. The last half of this chapter is devoted to describing a CAEN-based solution in detail.

We believe that change in computer architecture has historically been wrought by reversals of balance in the overall system structure. For example, we

have seen in Section 3.3.4 that memory performance grows more slowly than processor performance. At the point where the two performance curves intersected and processors overtook memories in speed, a change of balance occurred. This change warranted the introduction of caches. We will call such points *crossover points*.

Moore's law has been equated with a guaranteed stream of good news, bringing higher speeds and more hardware resources in each new hardware generation. However, if we extrapolate the technology graphs using the historical growth rate, we rapidly approach some important crossover points which spell trouble for conventional architectures.

3.4.1 Billion Gate Devices: Complexity Becomes Unmanageable

According to Moore's law, the feature size of CMOS integrated circuits decreases exponentially with time: every three years designers have four times more transistors available in the same area. In addition to the miniaturization of the basic components, the historic trend has been toward a steady increase in silicon-die size. Current microprocessors already have 30 million transistors and, in a few years, will reach 1 billion. Nanotechnology promises to push this number even higher, to 10^{10} gates per square centimeter [CWB+99, GB01]. The wealth of resources has enabled designers to create more and more complex circuits, harnessing extra circuitry to exploit the parallelism in programs and to overcome the latency bottlenecks. The downside of such huge sizes is the enormous complexity of designing, testing, verifying, and debugging such circuits. Most processors today are shipped with bugs [Col].

3.4.2 Low Reliability and Yield

When feature sizes become on the order of a few molecules, minor imperfections in the manufacturing process, random thermal fluctuations, or even cosmic rays can invalidate a circuit temporarily or permanently. Although some of the modern circuits contain error detection and correction capabilities (see Section 4), these solutions are still not universally applied to all parts of processor design.

3.4.3 Skyrocketing Cost of CMOS Manufacturing

The cost of manufacturing an integrated circuit can be separated into two components: the cost of the manufacturing plant and the non-recurring engineering (NRE) cost, which is on a per-design basis. The NRE cost includes testing and verification costs.

Moore's "second law" describes the exponential increase of the cost of a manufacturing plant with time. A state-of-the-art installation now costs more than 3 billion dollars. This cost comes mostly from the very precise mechanical and optical lithographic devices and masks. Another problem compounding cost is the yield, as devices with very small features are more prone to defects.

3.4.4 Diminishing Returns in the Exploitation of ILP

Only a small fraction of the area of modern microprocessors is dedicated to actual computational units. Upon close scrutiny, the only parts of the microprocessor which do actual computation are the functional units; on Pentium 4, for example, these make up less than 10% of the entire chip surface. All other structures on the chip only store and move information, they do no actual computation.

There is a tension in the design of the processor pipeline: on one hand, designers add enough resources to support program regions that can use them (with high ILP, unpredictable branches, etc.); on the other hand, most of the time these resources switch idly. For example, 4-wide issue processors seldom can sustain 1.5 instructions per cycle [HP96].

Processors cannot exploit ILP behind the limited issue window[9] [CG01]. However, the complexity of superscalar processors increases quickly with the

[9] The issue window is a set of consecutive instructions considered by the processor to be executed in parallel. A processor will not launch in execution instructions outside of the issue window, even if they are independent.

size of the issue window, with some hardware structures growing quadratically.

3.4.5 Power Consumption

The power consumed by a CPU cannot be thermally dissipated in an efficient way. Despite the fact that the supply voltage is decreasing for each chip generation, the total power consumption is skyrocketing. Modern CPUs have reached more than 100 W, exacerbating the difficulty of cooling [Aza00] and of supplying power (especially for mobile devices). More than half of the power is consumed just by the clock signal, which spans the whole chip surface [Man95]. Power density, which impacts directly the cooling efficiency, also grows with each generation.

3.4.6 Global Signals

The time between two clock ticks is not enough for a signal to cross the whole chip. A major problem engendered by the quickly increasing clock rate is that the electromagnetic signal does not have enough time to cross many levels of logic. Deeper pipelining is a partial solution but provides diminishing returns due to the overhead of the pipeline registers (the time for the information to cross the synchronizing latches starts to dominate computation time) [KS86].

Moreover, many structures on a modern CPU require global signals or very long wires (e.g., the forwarding paths on the pipeline, the multi-ported register files, the instruction wake-up logic). Wires connecting different modules tend to remain relatively constant in absolute size from one architectural generation to the next [HMH01, AHKB00], so they will be unable to function at higher clock rates.

4 Reliability in Computer System Architecture

4.1 Introduction

In this section, we discuss computer system architecture from the standpoint of reliability. Our treatment will be neither comprehensive nor formal; we mostly use examples to show how reliability considerations influence the design of computer systems.

The main topic of reliability theory is the construction of reliable systems out of unreliable components. If the correct behavior of a system as a whole relies on the correct operation of all its components, the reliability of the system decreases exponentially with the number of components. Without special measures designed to increase the reliability of the ensemble, no complex system would ever work.

Computers are extremely complex systems: a modern microprocessor contains more than 30 million transistors, while a 16 megabyte memory has more than 100 million. Computers themselves are aggregated in even more complex constructions, such as computer networks. The Internet is comprised of more than 125 million computers.

The main tool in the design of the complex systems is *abstraction*: high-level functionality is composed of low-level behaviors. The emerging high-level functionality is regarded as atomic for the construction of yet higher layers. This process is analogous to the use of theorems and lemmas in mathematical proofs: once a theorem is proven, it can be used wherever its conditions are satisfied.

Here we look at the interplay between layering and reliability. Layers can be designed which:

- enhance the reliability, offering the appearance of a reliable mechanism constructed of less reliable components. The main technique used for this purpose is *redundancy* in computation or storage of information. This approach is most frequently used in computer system design. Section 4.3 is devoted to examples of this type.

- expose the unreliability to the upper layers, allowing them to address the imperfections. The advantage of such a scheme is that it may be less expensive to implement. The higher layers have more knowledge about the environment and can then implement only as much reliability as is required for dealing with the errors likely to occur.

The architecture of the Internet is based on this paradigm, as described in Section 4.5.

- partition a system into (mostly) independent parts, which operate in isolation from one another. The effect of partitioning is the isolation of faults (fault containment) in parts of the system, leaving parts unaffected by the fault in working order. This paradigm is used in the design of operating systems and computer clusters.

The manufacturing of very large scale integrated (VLSI) circuits has achieved extraordinary reliability for the basic computer components. This enables computer engineers to treat most layers of a computer as perfect and to concentrate on synthesizing high-level functionality, without regard to reliability. Special classes of applications which require extremely high reliability (e.g., air-traffic control systems, nuclear power plant control) force system architects to introduce additional fault-tolerance providing layers. In general, increasing reliability comes at a cost. Such cost should be weighed against the benefit provided by the more reliable computer system.

The use of electronic nanotechnology as a basic component of computer systems will have a huge impact on their design methodologies. One of the factors that will dominantly alter computer system design is the intrinsic unreliability of chemically assembled components. Nanodevices will provide a building block with totally different reliability properties than the ones traditionally used by computer engineers. This is due to the uncontrolled nature of self-assembly,[10] along with the minute size of the nanodevices, which makes them very sensitive to thermodynamic fluctuations and high-energy particles. In this chapter, we argue that alternative means of increasing system reliability will be required for electronic nanotechnology.

4.2 Reliability

This section introduces some basic terminology.

[10] Relative to optical lithography, which is the technology used for the construction of VLSI integrated circuits today.

4.2.1 Definition

While this discussion is informal, we begin with a precise definition of the notion of reliability. We quote from [VSS87]:

> The term "reliability" has a dual meaning in modern technical usage. In the broad sense, it refers to a wide range of issues relating to the design of large engineering systems that are required to work well for specified periods of time. It often includes descriptors such as quality and dependability and is interpreted as a qualitative measure of how a system matches the specifications and expectations of a user. In a narrower sense, the reliability is a measure denoting the probability of the operational success of an item under consideration. [...]
>
> **Definition** The reliability, $R(t)$, of an item (a component or a system) is defined as the probability that, when operating under stated environmental conditions, it will perform its intended function adequately in the specified interval of time $[0, t)$.

Note the probabilistic nature of the definition of reliability. We should realize that there are no systems with perfect ($R(t) = 1$) reliability. A more appropriate notion is "good enough" systems, whose reliability is satisfactory for the intended usage and under specified budgetary constraints. We will see various methods that can be used to boost the reliability. However, these methods are not free and the systems built using them are reliable but not perfect. Ideally a system should be no more reliable than needed, balancing cost with benefits.

4.2.2 Redundancy

The key ingredient for building reliable systems is redundancy. We can distinguish between two types of redundancy, spatial and temporal (see Fig. 18).

Spatial redundancy uses more devices than strictly necessary to implement certain functionality.

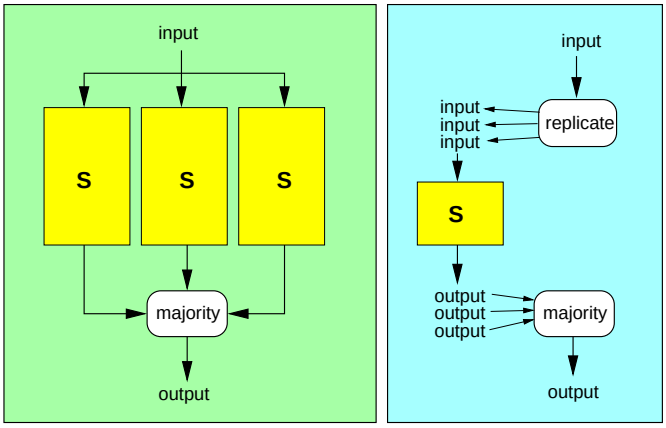


Figure 18: Spatial (left) and temporal (right) redundancy. Spatial redundancy replicates the computing element **S**, while temporal redundancy repeats the computation.

The additional devices are used to carry redundant computations whose results are cross-checked. If the reliability of each part is above a minimum threshold, the overall system will be more reliable than any of its parts. Using even larger amounts of redundancy allows a system not only to detect failures, but also to mask them.

Temporal redundancy involves the repeated use of a device for the same computation, followed by a comparison of all the produced results.

Temporal redundancy can only be used to cope with transient faults. Strong spatial redundancy, which enables error-correction (masking), can be used to tolerate the effect of permanent faults. We can expect the number of both permanent and transient faults to increase in future systems, due to the miniaturization of the basic electronic components. Small components use reduced electric charges to represent logic values and therefore are more sensitive to thermodynamic fluctuations, alpha particle radiation from the environment or even the chip package, or high-energy cosmic gamma rays.

4.2.3 Transient and Permanent Faults

According to their duration, we classify faults of a system into two broad categories:

Transient faults occur when the system temporarily malfunctions but is not permanently damaged. In today's computer systems, the vast majority of faults are transient in nature [SS92].

Permanent faults happen at some point and never go away, included in this category are manufacturing defects. We explicitly allow the use of partially functional components (i.e., having defects which affect only parts of the component) for building systems.

4.2.4 Reliability Cost and Balanced Systems

When designing a complex system it is important to balance the reliability of the various parts. For example, if both memory and processor must be fault-free for a system to be functional, it is wasteful to use a memory with a much higher reliability than the processor: the system will break down frequently from processor failures, so the increased memory reliability does not contribute substantially to the reliability of the system.

A more complete picture should factor in the system maintenance cost: what is the cost of the system downtime required for replacing the failed parts? Using an overly reliable part increases system costs. Using parts with high failure rates requires excessive maintenance costs. The context of use and the envi-

ronment of the system dictate how reliable it should be. For example, in safety-critical applications the cost of downtime is very large, so it makes sense to invest in extremely reliable components.

4.3 Increasing Reliability

In this section, we illustrate several techniques used for building reliable computer systems. Systems that recognize the inevitability of faults, and are designed to cope with them, are called *fault tolerant*.

4.3.1 Voting and Triple Modular Redundancy

A simple but expensive approach to fault tolerance is the complete replication of the computing system. Duplication and comparison of the results of the two copies permits the discovery of faults that occur in just one of the computing elements.

Triplication together with a majority vote can be used to recover the correct result in the presence of a faulty component. Triple modular redundancy (TMR) was the first fault-tolerance scheme. It was introduced in 1956 by John von Neumann [vN56]. It is sometimes used in computing systems for which maintenance is impossible, such as on-board computers for space probes. If each component has a sufficiently high reliability (above 50%), the result of the majority is more reliable than each individual component. The voting subsystem, which selects the result computed by a majority of the components, can itself be a single point of failure. Schemes have been devised which use replicated voters.

Voting can be naturally generalized to using n identical computations in parallel and a majority vote. Another generalization, which is more robust against permanent faults, disregards the vote of components after they produce a wrong result.

4.3.2 Error-Detecting and Error-Correcting Codes

Duplication can be used to *detect* wrong results, while triple modular redundancy can *recover* from one faulty computational element and voting methods permit operation after multiple faults. The cost of these schemes is considerable: duplication requires a complete replication of all critical computational elements, while TMR and voting have an effective resource utilization of less than 33%.

It is worthwhile to explore if we can achieve the same degree of robustness using fewer resources. The pioneering work of Claude Shannon [Sha48] and Richard Hamming [Ham50] in the 1940s has shown that we can indeed do better.

Let us consider the duplication method of fault tolerance. To make matters concrete, we start with the premise that we want to store a piece of data in a safe way. We assume that the data is encoded in binary and that a data word has n bits. If we just store two identical copies of the data (i.e., $2n$ total bits) what kind of robustness do we obtain? Let us assume that one bit is damaged, being flipped by noise. In this case, the values of the bit in the two copies will differ, so we will detect the fault. However, we have no way to discern which of the two values is the correct one. Duplication allows us to perform error detection, but not error-correction. Moreover, some two-bit failures can pass undetected by this system, if they occur in the same bit position in both copies.

Intuitively, this happens because the bits in the representation are rather fragile: each stored bit encodes information coming from exactly one original bit. We can build more robust encoding schemes if we "mix" the input bits and we use each stored bit to encode multiple input bits. In this case we may be able to detect or correct errors in a stored bit by extracting the lost information from other stored bits.

To obtain error resilience we need to add redundancy to the encoding. This is accomplished by encoding n-bit data using m bits, where $m > n$. The larger m, the more resilient the code can be. We call the m-bit encodings "code words." Note that not all m-bit words are code words.

There are many coding methods, each with different properties, suitable under different assumptions about the noise in the environment, the independence of the errors, and the complexity of the encoding and decoding algorithms. The interested reader can

learn more about this topic in the following resource [LC83].

In general, error-detecting methods are used when the information can be recovered from other sources. For example, in data transmission networks, the sent data is only encoded using error-detecting codes because if an error occurs the information can be resent by the source. When there is only one copy of the data, it must be protected with error-correcting codes. Such codes are used for example in computer memories and disk storage systems.

Main Memory If you have ever needed to upgrade the memory of a PC, you probably have faced the dilemma of the kind of memory to buy. Three main types of memory chips are sold today: unprotected, parity-checked, and error-correcting code (ECC) memory. These types of memory differ greatly in reliability.

Unprotected memory uses one bit of storage for each bit of information. It offers no protection whatsoever against transient or permanent faults and it essentially relies on the reliability of the underlying hardware. However, main memory capacity has grown exponentially over time, and the amount of storage in a computer today makes the likelihood of faults a much more probable event now than 10 years ago.

Parity-checked memory uses a very simple method to detect one bit errors.[11] It works by storing for each eight bits of data a code word of nine bits. The ninth bit is set such that each code word has an even number of "1" bits, hence the term "parity." Whenever data is read from memory, the hardware automatically checks the parity. When the parity is not even, a fault has been detected, and an exception is triggered. The software will handle this appropriately. A common course of action is to terminate the program using that piece of data (because there is no way the data can be recovered) and to mark that particular memory region as faulty, which will prevent further usage. Parity checking is simple and can be done quickly, so it does not slow down the memory operation.

ECC memory uses a more sophisticated encoding scheme that allows automatic correction of any 1-bit error occurring in a 64-bit word. For this purpose, the memory encodes 64 bits of data using a 72-bit code word. The overhead of the scheme is therefore the same as for parity-checked memory (9/8 = 72/64), but it is more robust than parity-checked memory since the latter can only detect a single fault in every 8 bits, while ECC can actually correct a single fault and detect 2 faults in every 64 bits. On each memory access the read data is brought to the closest code word. The code word is next decoded to obtain the original information. These operations are substantially more complex than a simple parity check. Therefore, ECC memory incurs a slight performance penalty compared to the other two types of memory.

Disks The most common support for persistent information storage is the disk. While disks come in many types (floppy disks, hard disks, removable disks, optical disks, compact disks, etc.), the facts presented in this section apply to most of them.

Disks use two different spatial redundancy techniques simultaneously. These are necessary due to the high storage capacity and to the harsh environmental conditions under which disks operate: unlike solid-state devices, disks feature moving parts which are far more likely to exhibit failures. A close approximation of a disk system is a turntable, which has a rotating disk and a head that can choose a certain track on the disk through lateral "seek" movements.

Information on disks is stored in units called *sectors*. The size of a sector depends on the disk but is relatively large (compared to the storage unit for memories, which is a 72-bit word for PCs), on the order of a kilobyte. Data inside of a sector cannot be changed: if even a single bit is modified, the sec-

[11] This method will also detect any involving an odd number of bits.

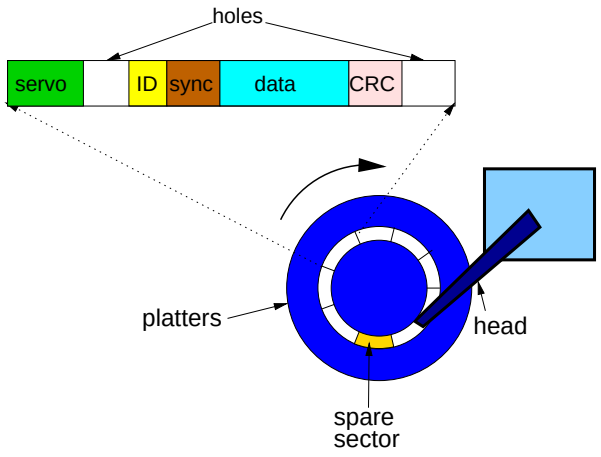


Figure 19: Redundant encoding of information on a disk: both error-detecting codes and spare sectors are used.

tor has to be rewritten entirely because mechanical imprecision makes it impossible to position the read-write head precisely enough for a single bit.

In addition to the stored data, a sector contains extra information to compensate for the imprecision of the mechanical parts and the high unreliability of the medium (see Fig. 19). For example, each sector has some servo information, which is used by software to check the head alignment and to synchronize the data transfer with sector boundaries. Traditionally each sector has an associated sector identifier, the usage of which will be described later. The data is stored following the servo and ID information, and is followed by an error-detecting code.[12] The codes used for disks are usually from a class called cyclic redundancy checks (CRCs).[13] Sectors are separated from one another with gaps, which give the head some freedom in setting the sector boundaries when a sector is rewritten. On rewrite, a sector may not start in the exact same spot as before due to imprecision in positioning.

Some events can damage sectors: for example, cutting the power of a computer while it is still operating can cause a crash of the heads on the disk surface, which causes physical damage to the magnetic layer used to store information. If this happens then, with very high probability, the checksum of the data residing on the damaged sector will not match.

To keep disks operational even after damage occurs, and also to allow some slack in the manufacturing quality, disks use a second type of spatial redundancy in the form of *spare* sectors. Regularly spaced on the disk surface are spare sectors that are normally unused under normal operation. When a sector is damaged, it is marked and replaced with a spare. The sector ID of the spare indicates which original is represented. Upon manufacture, disks are tested for surface defects and for each disk a *defect map* is created and stored within the disk system (either on the disk or in a nonvolatile memory). The defect map is used when formatting the disk to replace the defective sectors with spares. The defect map grows during the disk lifetime, as newly damaged sectors are added.

4.4 Formal Verification

Formal verification is an umbrella name for a variety of sophisticated techniques used for certifying the correctness mostly of hardware systems, but increasingly for software systems as well. Formal verification is used for finding bugs and in this sense can be seen as a reliability-enhancing technique.

The crux of formal verification methods is specifying precisely the behavior of the systems' components (in terms of mathematical formulas) and checking formally the properties of the overall systems. These properties are precisely expressed using some form of mathematical logic and are proven by a sequence of derivations.

One particular technology which we believe will play a very important role in future computer systems is *translation validation* [PSS98]. Using this technique, a compiler can produce proofs that the result of the compilation process faithfully implements the source program. Because modern computer architectures are extremely complex, they rely more and more on compilers. Translation validation can therefore be used to certify an essential building block of computer system architecture.

[12] More precisely, the data plus the following information form together a code word of an error-detecting code.

[13] A CRC summarizes the data using a *checksum*. The term "cyclic" comes from the property that the same checksum is obtained if the input data is permuted cyclically.

4.5 Exposing Unreliability to Higher Layers

The first part of this section was dedicated to a discussion of techniques used to enhance the reliability exposed to the upper architectural layers. We noted that hardware is much more reliable than the software layers because it is virtually fault-free.

In this section we discuss the opposite approach used in two very successful systems. These systems start from the assumption that the underlying layers are essentially unreliable and that unreliability must be *exposed* to the upper layers.

This seemingly paradoxical design can be defended by several arguments:

- The cost of providing reliability may simply be too high. Moreover, the cost grows with the number of abstraction layers. The overhead may be unacceptable for some classes of applications.
- The upper layers may use their own fault-tolerance scheme. The designer faces a trade-off between the probability of failure and the effectiveness of a fault-tolerance scheme. Each reliability scheme we present works under certain assumptions about the environment. For instance, the memory parity scheme in Section 4.3.2 stores one bit of parity for each byte. This scheme works well as long as there are not multiple errors within a same byte (i.e., the error probability of a bit is relatively low). A lower layer may forgo providing a reliable appearance for the layer above if the latter uses an error-correction scheme powerful enough to withstand the combined error probability.
- In some cases, the upper layers need only a simple form of reliability, and not a perfect underlying substrate. For example, some interactive applications such as telephony would rather have the data delivered timely than perfectly: a missing piece of data generates a glitch at the receiver, which is often tolerable by the human ear. By contrast, the late arrival of the complete data is of little value for creating the illusion of an interactive conversation.

4.5.1 Internet Protocol

The Internet is a computer network, designed initially to link military computer networks. One of the design goals of the Internet was to make it robust enough to withstand substantial damage without service loss. The Internet has evolved into a highly successful commercial network spanning all continents, with over 125 million active computers.

The Internet was created more than a century after the deployment of the telephone network, therefore one would expect the Internet to build upon the principles developed during the construction of its antecessor. However, particularly with respect to reliability, nothing could be further from truth: the Internet architecture is almost the complete opposite of the telephone network.

In order to contrast the two networks, we first discuss some details on the architecture of the telephone networks.

Telephone Network Architecture The telephone network was designed to have an exceptional reliability. The downtime of a telephone switching exchange must be under three minutes per year. Only under truly exceptional circumstances is a conversation already initiated interrupted due to a network failure. The phone network will let a conversation proceed only when it has secured all the required resources to timely transmit the voice signals between the parties. Strict standards dictate the length of time allocated for the connection negotiation phase, which establishes an end-to-end circuit through all the intermediary telephone exchanges. The failure to secure all necessary resources is signaled to the end user through a busy tone. Sophisticated capacity planning based on detailed statistics about phone user behavior are used to size the carrying and switching devices, which results in a very low probability of a busy tone due to insufficient network resources.

A crucial factor guaranteeing the quality of the phone connection is the allocation of all necessary resources before the call is connected. These resources are preallocated and reserved for the whole

duration of the connection [Kes97]. Based on the number dialed, the path connecting the source and destination is calculated using carefully precomputed routing tables. Each exchange negotiates with the next one on the path using a sophisticated signaling protocol carried on special circuits, allocating bandwidth and switching capacity for the new call. After all point-to-point connections are successfully established, a ring tone is generated. Once the conversation begins, the voice signal is sampled and digitized at the first telephone exchange. Each bit of data has preallocated time slots in each of the trunks it will traverse. The data bits traverse all trunks in the order they were generated and arrive at the destination at the right time to enable the recreation of the analog voice signal. For each incoming bit, switching tables maintained in each exchange indicate the outgoing circuit and precise time slot where the bit should be steered. Once a connection has been established, there is a very high probability that the data entering the network will emerge at the other end in the right order and with the same delay for each bit. On disconnection, the signaling protocol tears down the resources allocated during the call.

The Internet Architecture One of the aspects where the Internet philosophy fundamentally differs from the telephone network is reliability. Not only is there no guarantee about the latency taken to transit the network, but there is no guarantee that the data will not be lost or corrupted during transit. The end users of the Internet get a very weak service definition, which could be stated as: "You put data in the network tagged with some destination address. The network will try to deliver your data there."

The way data travels in the Internet is completely different from the telephone network: data is divided into packets that are injected in the network in the order they are generated. Each packet may travel a different route to the other end point. Some packets may be lost, some may be duplicated, and the receiver may receive the packets in the wrong order and perhaps even fragmented into smaller packets [Kes97].

The packets are moved by the Internet protocol (IP). IP works roughly as follows (see Fig. 20): when an intermediate computer (a router) receives a data packet, it reads the destination address contained within. Next the router makes an informed guess about the direction the packet should travel (i.e., the directly connected neighbor closest to the final destination). The packet is then forwarded to that neighbor.

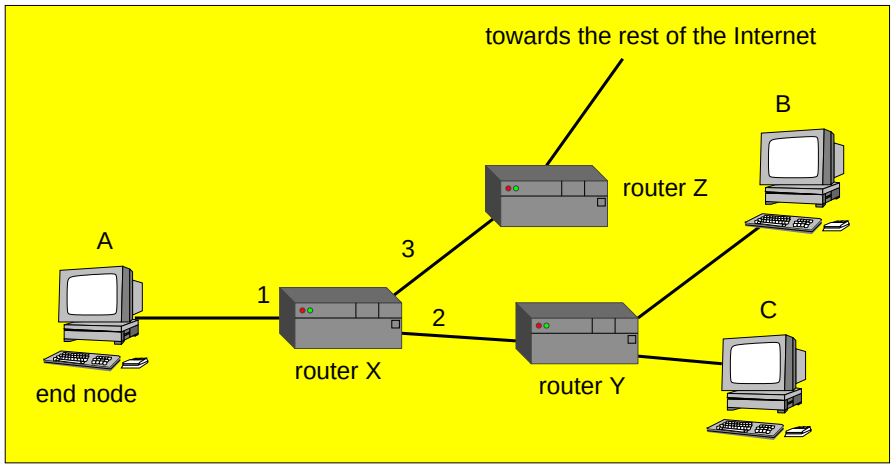
When a router receives data faster than it can send on the output links, it buffers them in its internal memory for later processing. When the internal memory becomes full, the router simply starts to discard the packets. Such dropped packets constitute the major source of performance degradation in the Internet.

The structure of the Internet changes widely from day to day and from hour to hour, as new computers get connected, new users dial-up, accidents disrupt the connectivity of some networks, and new communication links are installed or upgraded. Routers constantly exchange information about their local view of the network: each machine tells its neighbors the information it currently has about the topology. Constant information exchange leads to distribution of network topology information to all participating routers. Given this infrastructure, it is amazing that the Internet works at all and that information flows through it without corruption. How is this possible?

The answer comes from two key aspects of the Internet architecture: the core network is redundant and self-healing, while the upper layers are designed to tolerate information loss.

The core network achieves reliability using the following ingredients:

- **Rich, uniform, and redundant core:** The Internet "backbone" is not a simple linear structure; commonly between any pair of nodes there are multiple disjoint communication paths. Moreover, there is no single point of failure whose destruction can disconnect the network. An important factor is that all nodes in the network are more or less functionally equivalent: there is no privileged node on whose health the whole system depends.



Router X

Destination	Use wire
A	1
B, C	2
everything	3

Figure 20: **Left**: In the Internet there are end nodes (which initiate the communication) and routers (which only move data). **Right**: Routers use routing tables to decide which interface to use for each destination address. This example shows router X's table.

- **Statelessness:** The routers in the Internet do not store information about the data traffic flowing through them. (In contrast, a telephone exchange stores much information about all phone connections that traverse it.) A router receives a packet, computes the best output link, and forwards the packet. The internal state of the router after forwarding is the same as before the arrival of the packet. The lack of stored state enables the network to withstand the crash of routers (i.e., a crash does not cause loss of information which cannot be recovered through other means).
- **Self-testing and healing:** The routing protocol constantly exchanges information about the topology of the network. In the event of network outages, this protocol quickly propagates information about backup routes, allowing communication to resume.

User applications use a robust communication protocol that makes use of temporal redundancy to tolerate errors in the underlying network. The transmission control protocol (TCP) is a communication protocol executed only by the end nodes participating in communication. This protocol ensures that all packets are delivered in the order they were sent, without gaps or duplicates. TCP achieves this feat by using the following ingredients:

- numbering the sent packets
- acknowledging packet reception
- using timers to detect packets which received no acknowledgements for a long time
- retransmitting packets deemed lost in the network

Because acknowledgments are sent using ordinary packets themselves, they may be lost too. Lost acknowledgments cause the injection of duplicate packets in the network by the source which retransmits the packets not acknowledged.

The whole Internet is built on the unreliable IP core: not only are data and acknowledgment packets sent unreliably, but also the control communication between routers (describing network topology) and the network maintenance traffic use the same unreliable forwarding mechanism.

Despite its apparent weak structure, the Internet is a formidable competitor to other specialized media distribution networks: radio, television, and tele-

phone. The cost of voice transmission (telephone-style) in the Internet (dubbed "voice over IP" or "VoIP") is much lower than in telephone networks. Many major phone carriers are already investing heavily in using VoIP in parts of their networks.

Why is the Internet more successful *despite* its unreliability?

The answer lies, at least in part, in the fact that the Internet does not offer a costly service (e.g., fault-free time sensitive communication primitives). Instead, it offers a cheap but useful service (e.g., best effort delivery of bits). The telephone network goes to great lengths to provide the reliable voice service but uses an extremely inflexible, specialized system and wastes a huge amount of resources. Basically, the reliability of the phone network comes at *a high cost*.

The Internet moves the fault tolerance problem to a higher layer, from IP to TCP. TCP provides fault tolerance perfectly adequate for many applications. TCP is executed only by the end hosts and not by the routers who relay the information. The complicated processing entailed by TCP does not place any burden on the core network, which scales to global sizes.

Applications that do not need the reliable data delivery offered by TCP can forgo using it. For instance Internet radio protocols use strong error-correcting codes and no retransmissions. Lost or misplaced packets are simply ignored. This is acceptable because the human end user *tolerates the low reliability* of the signal.

4.5.2 Teramac

In this section we present briefly Teramac, a computer system developed by researchers at Hewlett-Packard, which exhibits another unconventional approach to the problem of system reliability. This computer is built out of defective parts: more than 70% of the circuits composing it have some kind of fault. Despite this, the system works flawlessly. Many papers have described Teramac. One article which underlines the connections between this architecture and nanotechnology-based computing is [HKSW98].

It must be noted that Teramac deals only with permanent faults. Electronic nanotechnology circuits are very likely to exhibit a large number of such faults.

Teramac is built out of reconfigurable hardware devices, described extensively in Section 5. The reconfigurable hardware feature exploited by Teramac is the fact that the basic computational elements are essentially interchangeable.

Teramac implements circuits on a reconfigurable substrate. Much like how disks use spare sectors to replace the faulty ones (see Section 4.3.2), Teramac uses spare computational elements to replace the faulty ones. In Section 5, we describe the mechanism for accomplishing this after we present the main features of reconfigurable hardware and describe how such devices are programmed.

One of the main contributions of the Teramac project is showing that unreliability in hardware can be exposed and addressed by higher software layers, without incurring a large overhead. This is a complete paradigm shift in computer system design, which will very likely have many applications in the future.

4.6 Reliability and Electronic-Nanotechnology Computer Systems

4.6.1 Summary of Reliability Considerations

In this chapter we surveyed many ways in which reliability considerations impact the construction of computer systems.

Computer systems are built from a series of abstract layers, which offer increasingly powerful abstractions. Each layer has different reliability properties and uses different reliability-enhancing techniques. In general, the view exposed by the hardware to software is that of a practically perfect, fault-free substrate.

We have seen that reliability can be enhanced through the use of redundancy: spatial redundancy

uses more hardware components to carry supplementary computations, while temporal redundancy repeatedly computes results to ensure their robustness against transient faults.

While temporal redundancy is expected to reduce the overall system performance (expressed in number of computations per unit time), even spatial redundancy comes at a price: for example, we have seen that using powerful error-correcting codes for memories can cause performance degradations due to the complexity of the additional computation of the code.

One important message of this discussion is that reliability is a desirable feature but it comes at a certain cost, which must be taken into consideration by the designer. The designer faces a trade-off between the cost of reliability of each layer and the fault tolerance that upper layers can provide by themselves. The deep layering translates into a multiplicative effect of the overheads.

4.6.2 A New Computer System Architecture

The design of future computer systems, based on CAEN, will very likely be fundamentally constrained by the dramatically different reliability properties of the new medium. We can expect both very high permanent defect rates from the nondeterministic manufacturing process and a high rate of transient defects at the lowest layers, due to the thermodynamic fluctuations and the minute size of the currents carrying information in CAEN circuits. (Note that the latter issue, transient errors, is not unique to CAEN, but will be common in any technology which measures individual features in handfuls of nanometers.)

The use of CAEN devices will require a complete rethinking of the basic architecture of a computer system. In this section we sketch features of a plausible proposal. We speculate that by diminishing the cost paid for reliability we can reduce dramatically, even by orders of magnitude, the cost of complete computing systems. We advocate:

- removing the illusion of perfect reliability from the hardware layer, exposing the imperfections of hardware to the software, and dealing with the imprefections on as needed basis
- using as a basic building-block reconfigurable hardware circuits (see Section 5), whose configuration and reliability are completely handled by software programs
- using defect-mapping and spares for dealing with manufacturing defects
- using computation in encoded spaces (i.e., computing on data encoded using error-correcting codes) for transient and permanent fault-tolerance
- using formal verification techniques, most notably of translation validation, to ensure that the compilers correctly translate programs into executables or configurations
- offloading most of the computation from the microprocessor and instead using a reconfigurable hardware substrate on which application-specific circuits are synthesized by compilers. Compilers will generate hardware configurations matching the application's needs and reliability requirements

decribe program V. configuration somewhere

5 Reconfigurable Hardware

In Section 3 we briefly presented the main paradigms exploited in the architecture of modern computer systems. We have seen that fast-evolving technology constantly changes the balance between the computer system components, which in turn requires constant redesign and tuning. We have concluded that computer architects face several difficult problems, engendered by the enormous design complexity, increasing power consumption, and limitations of the materials and manufacturing processes.

In this section we describe a computational paradigm which has accumulated momentum during

the last decade and may ultimately solve some of these problems. This paradigm is embodied in the reconfigurable hardware (RH) circuits [CH99]. RH is an intermediate model of computation with features between special-purpose hardware circuits and microprocessors. Its computational performance is close to hardware, yet its flexibility approaches that of a processor, because RH is programmable.

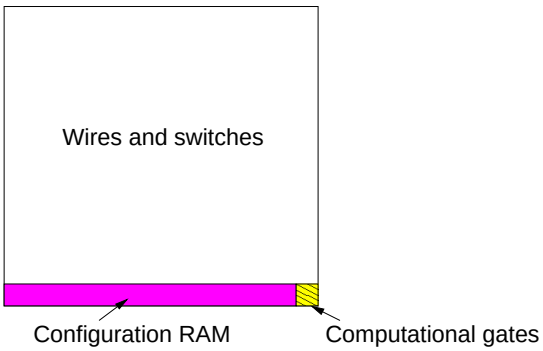


Figure 22: Relative area occupied by the components of a reconfigurable hardware device. The actual computational devices take just a small fraction of the chip (around 1%).

5.1 RH Circuits

As described in Section 2.2, ordinary digital circuits are composed from simple computational elements called *logic gates*, connected by wires. The logic gates are themselves assembled from transistors. Each logic gate performs computations on one-bit values.

RH devices have a similar structure. However, the logic gates of an RH device do not have *a priori* fixed functionality and the wires do not connect specific pairs of gates but are laid out as in a general network. Each gate is *configurable*, which means that it can be programmed to perform a particular one-bit computation. Likewise, the connectivity of the wires can be changed by programming switches that lie at the intersections of the wires.

Each logic gate and each switch has a small associated memory that is used to store the configuration. By loading different data in each memory, the gate functionalities and wire connections are changed (see Fig. 21a). Because configuration can be changed repeatedly, these devices are called "reconfigurable." A second interconnection network is used to send configuration information to each programmable element.

RH circuits ordinarily have a regular structure and are composed of a simple tile replicated many times along the horizontal and vertical direction. This symmetry has an important role in some RH applications.

RH is as powerful as standard hardware because any circuit that can be built in regular hardware can also be synthesized using RH. The price paid by RH for its flexibility is relative inefficiency: the programmable gates and switches, and the associated configuration memories, are much larger than the custom gates and wires in a normal circuit.

The limiting resources on RH are not the gates, but the wires, because it is not known beforehand which connection will be made. The chip must have enough wires for the worst case [DeH99]. Any particular implemented circuit ordinarily uses just a fraction of the available wires. To handle the peak demand, RH circuits are oversupplied with wires, which ultimately take most of the silicon resources: normally 90% of the chip area is devoted only to wires and their configurable switches. Fig. 22 illustrates the on-chip area dedicated to the wiring, configuration memory, and actual computational devices on a typical RH chip.

As a consequence, a circuit implemented in RH is larger (in area) than the corresponding custom hardware circuit. Because RH integrated circuits have approximately the same absolute size limitations as ordinary digital circuits, large circuits do not fit into one RH chip. However, today's RH devices have capacities on the order of a few million gates, an amount greater than advanced microprocessors from a decade ago.

Besides the density handicap, RH devices also have a speed disadvantage when compared to custom hardware: electrical signals in custom hardware travel on simple wires between two gates; in RH signals frequently have to cross multiple switches from a source to a destination. The switches increase

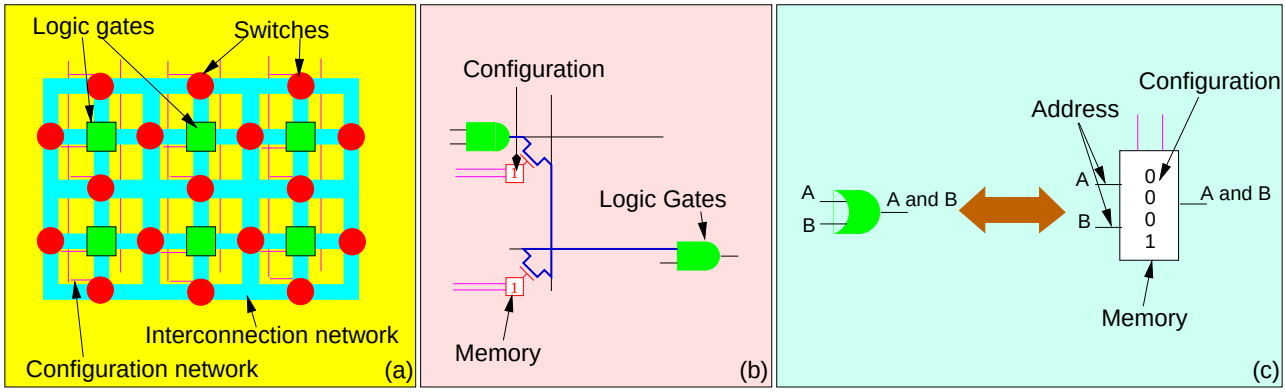
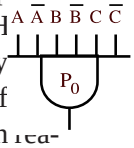


Figure 21: (a) Reconfigurable hardware contains programmable logic gates and an interconnection network. (b) The connections between the logic gates can be set up by configuring the switches in the interconnection network. Each switch is controlled by a one-bit memory. (c) Programmable logic gates can be implemented as look-up tables using small memories. In this example, the contents of the memory indicate that the gate performs the AND computation.

the power consumption and decrease the propag
speed of the signal.

The overwhelming majority of digital circuit
synchronous (i.e., use a clock signal to coordi
the action of the various computational units sp
on the chip surface). This is also true of RH
vices, although the peak clock frequency they
typically use effectively is approximately one-fif
one-tenth of a microprocessors' clock. The main rea-
son for this disparity is that the electrical signals must cross several switches on almost any path.



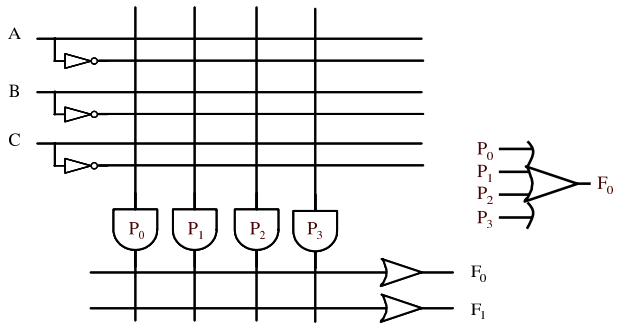


Figure 23: A schematic of a PLA which can implement a subset of all logical formulas of four inputs. The single line into each AND gate represents six wires coming from the three inputs and their complements. The AND gate to the left of the figure shows all six input lines going into the P_0 AND gate. The single line into each OR gate represents four wires, one each from the AND gates. The OR gate to the right shows all four input lines for the F_0 output.

5.2 Circuit Structures

The implementation of RH circuits described above is just one possible method of implementing programmable logic. Here we describe three other approaches that could be used. The simplest of the methods is to use a single monolithic memory. A memory with n locations (i.e., $\log_2 n$ address lines of m bit words can implement any m functions of $\log_2 n$ inputs). In Fig. 21c, a memory of four locations of one-bit words is shown. It implements one function of two inputs. If we increase the word width we can add an additional truth table. If we increase the number of words, we can increase the number of inputs. One of the primary reasons RH devices, like field-programmable gate arrays (FPGAs), are composed of many small memories (instead of one large memory) is the exponential growth in the size of the memory in terms of the number of inputs to the function.

Historically, programmable logic devices were implemented more like monolithic memories than FPGAs. To avoid the scaling problems with a single memory, another common implementation approach is to use a programmable logic array (PLA). A PLA directly implements two-level logic functions. A logical function can be written in many dif-

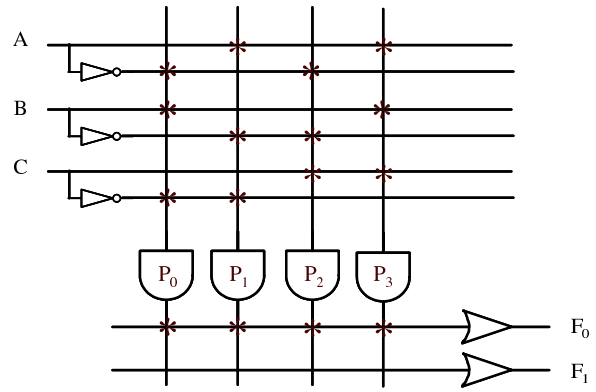


Figure 24: Example programming of a PLA to implement the sum output for a full-adder. This example uses only one of the two output functions.

ferent but logically equivalent ways. One of the most common is to represent a logical function in either a sum of products (SOP) form or a product of sums (POS) form. A sum of products form of a function is constructed by summing a set of product terms. Each product term is the AND of some of the inputs or their complements. Each sum term is the OR of some of the product terms. For example, the SOP representation of the sum output of the full-adder in Fig. 3 is sum $= \overline{a}b\overline{c} + a\overline{b}\overline{c} + \overline{a}\overline{b}c + abc$. The idea behind a PLA is to directly implement SOP by having a programmable product plane that feeds into a programmable sum plane.

Fig. 23 shows an example PLA which can compute two different functions (one each on the horizontal lines) of four product terms (computed on the vertical lines). The central figure is a common form of abbreviation for a PLA. It avoids drawing all the lines that enter each AND and OR gate. On either side we show a single example of the AND and OR gates in their unprogrammed form. Programming a PLA involved removing some of the connections into the AND or the OR gate. The PLA shown in the figure can implement only a subset of all possible three input functions. For only three inputs the PLA is considerably more expensive to implement than a look-up table (in this case it would be an eight-word memory of two bits per word). However, PLAs can quite efficiently implement functions of 16 inputs, whereas such a memory would be highly inefficient.

Fig. 24 shows how the sum output of the full-adder might be implemented in a PLA. We represent each wire that should remain connected to the AND or OR gate with a star on the intersection of the signal and the wire going into the gate.

While PLAs are more efficient than single memories, they still have significant overhead to support the programming of the AND and OR planes. Programmable array logic (PAL) was introduced to reduce this overhead. In a PAL, the inputs to the OR-plane are fixed so that only the AND-plane can be programmed. This restricts the outputs to the sum of a fixed set of product terms. This decrease in flexibility is offset by a large reduction in the area of the device.

5.3 Using RH

The RH industry had a revenue of 2.6 billion dollars in 1999 and its growth is accelerating. The same trends that affect the architecture of computer systems are shaping the use of RH hardware today. We are witnessing an important change in the role relegated to RH devices in computing systems.

5.3.1 Glue Logic

RH was originally developed as a flexible substitute for circuits used to "glue logic," connecting the custom hardware building blocks in a digital system. RH was used for *fast prototyping*: new circuits could be designed, tested, and deployed without the need for a manufacturing cycle. Even today, glue logic is the most important segment of the RH market. The prototypical RH device of this kind is called a field-programmable gate array (FPGA). The architecture of FPGAs is profoundly influenced by their usage: because they are used mostly for implementing "random" control logic, FPGAs favor one-bit logic elements. This can be contrasted to microprocessors, which are designed to process numeric data and therefore feature a wide *datapath* processing wide data values as atomic units.

5.3.2 Computational RH Devices

Feature shrinkage enables the manufacturing of RH devices with more and more computational elements. When large enough, they can accommodate computational tasks usually relegated to microprocessors. Intense research has been conducted in the last ten years on this subject, leading to many proposals for integrating RH devices into computing systems [Har01]. RH devices have an enormous potential as computational elements. They can successfully complement—and even supplant—the microprocessor as the main computational core of a computing system [BMBG02]. In the remainder of this text we will address RH devices only as computational devices.

5.3.3 Computational Applications for RH

RH as a computational device has drawn attention due to some impressive early results [Hau98]. For some application areas, RH still holds performance records, even occasionally when compared to custom hardware solutions. Even when they do not hold the performance crown, RH devices are substantially cheaper to manufacture and program than other computing systems with large peak performance, which feature either custom hardware or a large number of processors.

In Section 5.5, we investigate the source of the RH advantages. Here we will list some of the successes of RH computational engines.

Computational applications exhibiting extreme performance on RH hardware include: text search and match, including DNA string search; encryption and decryption; digital signal processing, such as filtering; and still and video image processing. These applications are all "streaming"-type applications in which the same operation is applied to vast quantities of data repeatedly. Such applications exhibit substantial amounts of data parallelism (i.e., many pieces of data can be processed simultaneously). The speed-ups obtained using RH systems (compared to conventional computing systems) range from one to several orders of magnitude.

Recently, problems with more complex structures have been substantially accelerated, such as graph searches and satisfiability of Boolean formulas.[14] Performance increases of lesser magnitudes have been reported even for less regular applications.

5.4 Designing RH Circuits

The tool chains used today for configuring RH devices reflect their inheritance from the random logic domain: the RH programming tools are mostly borrowed from the hardware design tools. These tools are substantially different in nature from the regular software design tools because they have different optimization criteria: hardware design tools tend to have very long compile times, extremely complex optimization phases, and very stringent testing and verification constraints. Much of the effort of the hardware design tools is directed toward addressing the constraints of the physical world: examples include laying out wires with few crossings and working with the actual geometry of the devices and electrical constraints (e.g., electrical wire resistance).

In contrast, software tools are more interactive: making small changes in software programs and rebuilding the whole executable in a matter of seconds is customary. Unlike hardware, software programs are frequently upgraded in the field.

The major steps in programming RH devices follow quite closely the design of digital circuits (see Fig. 25): the desired functionality is described using a program written in a hardware description language (HDL). Using this representation, the circuit can be simulated using computers for testing and debugging. Next, the program is processed by a series of sophisticated software tools, computer-aided design (CAD) tools, which first optimize the circuit, then *place* the logic gates on the two-dimensional surface, and finally *route* the wires connecting the gates (see also Fig. 26). In the case of RH devices, placement associates each logic gate in the circuit with one of the programmable logic gates of the RH

[14] Satisfiability is the problem of deciding whether a given Boolean formula can be made "true" by some assignment to its variables. Many important practical problems can be reduced to satisfiability computations.

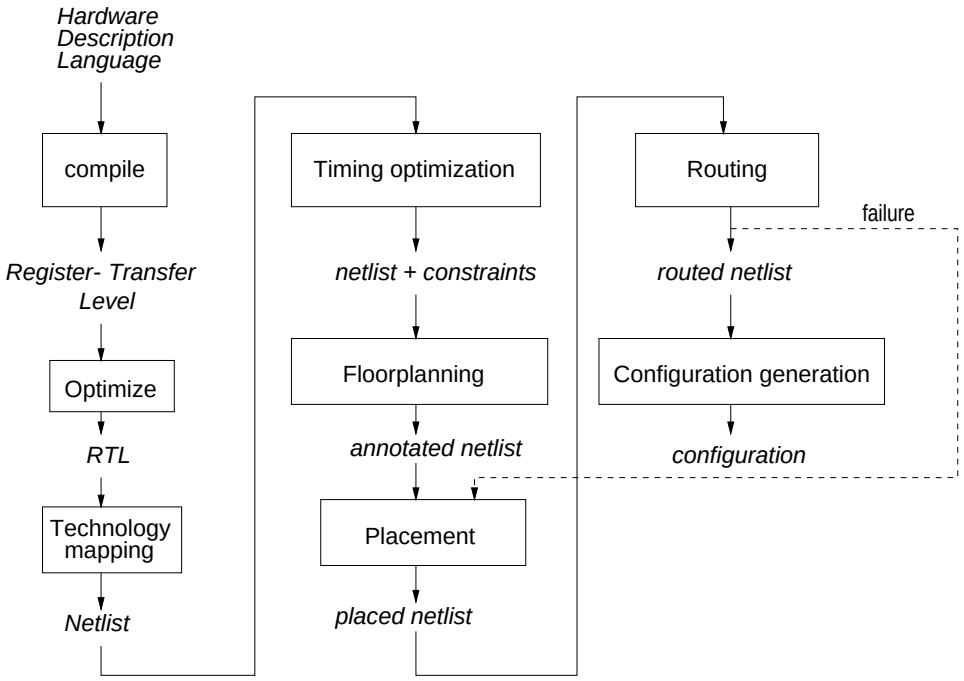


Figure 25: Compiling for RH devices is a complex process involving several complex optimization phases. Floorplanning, place and route must take into consideration the constraints of the chip geometry.

device, while routing selects wire segments and configures the switches to connect the gates linked in the implemented circuit.

RH CAD tools output a stream of bits, which constitute the configuration of each gate and switch. At run-time this configuration is loaded onto the chip. After configuration loading, the RH device starts behaving like the circuit whose functionality it implements. Using RH is therefore a two-stage process: loading the configuration and running the device. Because of the large number of gates and switches, the configuration of a chip tends to be very large. As a consequence, configuration loading is quite lengthy. Evaluating the performance of a RH device must always factor in this overhead, which is not present in either processors or custom hardware.

To shorten the configuration time, novel designs incorporate special features. As an example, some RH devices have local memories which can be used to cache several frequently used configurations [LCH00]. In this case, switching from one configuration to another can be achieved somewhat faster. Other designs can switch quickly between configurations; these are called "multicontext FPGAs" [DeH96]. Some other proposals allow only parts of the chip to be configured, while other parts compute, allowing the computation and configuration to overlap [GSM+99].

5.5 RH Advantages

In this section we discuss some of the advantages of RH devices over other hardware or software solutions. We also analyze the features of RH which enable it to exhibit very high computational performance. We defer to a later section (5.7) a discussion of reliability of RH.

Compared to custom hardware solutions, RH-based implementations have a shorter design and fabrication time and substantially lower costs. The fabrication time and costs advantages result from the reuse of the same integrated circuit: there is no need to physically manufacture new chips. Development and debugging are expedited, as the discovery of bugs results in configuration patches, which are gen-

erated by a chain of software tools and require no physical manufacturing.

The principal advantage of RH over pure software solutions is increased execution speed, at least for some types of applications. In examining these applications, we realize that they exploit four main RH qualities:

- **Parallelism:** These applications feature a large amount of intrinsic parallelism (i.e., many operations can be performed simultaneously). RH can exploit virtually an unbounded amount of parallelism. A microprocessor is outfitted by the designers with a certain number of functional units. Each of these units is unable to produce more than one fresh result in each clock cycle. The *computational bandwidth* of a processor is therefore bounded at the time of manufacture. Moreover, very rarely can a processor reach its peak performance because the parallelism available in the program rarely has the exact same profile as the available units. This is discussed in Section 3.4.4.

  In contrast, by using RH one can synthesize a virtually unbounded number of functional units.[15] For example, if the program contains 10 independent additions, one can build a configuration featuring 10 parallel adders. Not only can we build highly parallel computational engines, we can do so only after we know the application requirements—a luxury that a microarchitect cannot afford.

  Two main types of parallelism are successfully exploited by RH devices: parallelism between independent operations (multiple data items can be processed simultaneously) and pipeline parallelism, described in Sections 3.3.2 and 3.3.3. As an example, let us consider a program processing a digital movie, for instance, by decreasing the brightness and next thresholding (i.e., displaying in white everything above a certain intensity and the rest in black). In each movie frame the brightness of all pixels can be increased in parallel: this is parallelism between independent operations. Moreover, while one frame is brightened, the previous frame can be thresholded by using a set of different processing units. This is a form of pipeline parallelism.

- **No instruction issue:** Unlike a microprocessor which has to fetch new instructions continuously, the functionality of RH is hardwired within the circuit. The RH unit does not need to expend resources to fetch instructions from memory and decode them.

  Dispensing with instruction issue has several positive benefits. First, the problem of control dependencies[16] is eliminated or substantially reduced. Second, precious memory bandwidth is conserved because there is no need to bring instructions from the memory (the memory is connected with a narrow bus to the processor, on which there is contention for transmitting both data and instructions).

- **Unlimited internal bandwidth:** A subtle difference between a processor and an RH device is in the way they handle intermediate computation results. Processors have an internal array of scratch-pad storage elements called *registers*. Intermediate computation results are stored in the registers, where they can be quickly accessed. When a "final" result is produced, it is stored in memory. Like the number of functional units, the number of registers is hardwired from manufacturing. If the number of temporary values exceeds the number of registers, they must be *spilled* into memory, which is much slower to access. A scarce resource in processors is the number of *ports* into the register file. Data is read from or written to a register through a port: therefore, the number of ports determines the number of simultaneous read/write requests that can be honored by a register file. In other words, the bandwidth in

[15] Due to Moore's law, resource constraints in RH become less important with each new technological generation and will cease to constitute a real obstacle in the near future.

[16] We say that instruction B is control dependent on A if A decides whether B is executed or not. For example, branch instructions decide the next instruction to execute; the targets of a branch are thus control-dependent on the branch. Not knowing which way execution goes prevents the timely issuing of instructions, penalizing execution performance.

and out of the register file is essentially bounded by a hard limit—the number of ports.

In contrast, in RH devices the datapath is custom synthesized for each application. For instance, there is no need to use a *single* register file for all intermediate results. Because the data usage of the program is known, registers can be connected exactly to their producers and consumers, and nothing else. Moreover, an arbitrary number of registers can be created in the configuration because their space overhead is quite small. In RH, potentially unbounded register bandwidth accommodates the parallelism in the application under consideration, while the unbounded number of registers prevents costly memory spills.

- **Out-of-order execution:** While superscalar processors allow instructions to execute in orders different from the one indicated by the program, the opportunity to do so is actually quite limited:

  – First, only instructions within a limited "issue window" can be reordered.

  – To ensure correct execution, most superscalar processors constrain the instructions to also complete execution in program order.

  In RH implementations none of these constraints has to be obeyed. **explain why!**

Besides these major advantages of RH fabrics, there are three additional advantages which boost RH performance, although state-of-the-art compilation tools do not exploit them as effectively:

- **Custom widths:** The word size of a microprocessor is essentially fixed from manufacturing. When computations on narrower or wider data values are desired, programmers resort to word-size computations, which basically wastes most of the result. In contrast, RH devices allow complete flexibility for the width of computational units [BSWG00]. However, as we discuss in Section 5.6, there is a tension between efficiency of configuration size and logic density on one side and resource waste due to short data word size. For this reason, narrow computational units may not always be best in terms of performance.

- **Custom operations:** The functional units inside a microprocessor are restricted to a small set of general-purpose operations: simple integer arithmetic and logic, and bit manipulations. As media processing algorithms were standardized, processors started to incorporate special instructions designed especially to accelerate these programs [PWW97]. For example, virtually all modern processors have special instructions to aid with the encoding and decoding of digital video sequences. However, new applications and algorithms arise continuously and require special-purpose operations. In RH one can easily synthesize efficient implementations for operations which, when expressed in classic ISAs, require many instructions. As an example, consider an operation that permutes the bits of a word in a fixed order. In a normal ISA this requires at least three instructions per bit, for extracting, shifting, and inserting the bit. However, in an RH with bit-level operations the permutation can be implemented just by wire connections, without any computation.

- **Specialization:** This is applicable when some of the data processed by the algorithm changes slowly. As an example, consider an encryption algorithm with a given key: once the key is known, the algorithm will process a lot of data. The algorithm can therefore be *specialized* for a key [TG99]: once the key is known, an encryption circuit built especially for that particular key can be generated, which has the potential to operate much faster than a general-purpose encryption circuit. However, such specialization must generally be done at run-time, when data is known. This requires executing some of the configuration-generation process during program execution. This can be a costly process, which may offset the advantages of specialization.

5.6 RH Disadvantages

In this section we discuss the disadvantages of RH devices as compared to microprocessors. Some of the disadvantages are intrinsic to the nature of RH devices: for example, the need of the electrical signal to cross multiple switches for practically all point-to-point connections is a consequence of the generality of the routing fabric. Other disadvantages are "inherited" from the traditional usage of RH devices as glue logic circuits, for example, the cumbersome and slow programming methodology. These latter disadvantages will possibly be overcome by new research results.

We list disadvantages in order, beginning with the ones most likely to be easily overcome, and ending with the ones that seem fundamental.

- **Bit granularity:** The traditional usage of FPGAs for implementing random control logic has economically favored architectures oriented toward one-bit computations. Synthesizing wide arithmetic units from one-bit elements results in slow and large functional units (e.g., a custom eight-bit adder is much more compact than eight one-bit adders hooked together). This fact reflects a trade-off between flexibility and performance. If RH is to be used to accelerate general-purpose computation, most likely the balance has to be shifted more toward word-oriented devices [GSM+99, Har01].

  Closely related to the granularity of the computational elements is the flexibility of the routing resources. When RH is used for computation the full generality of a fine-grained interconnect is unneccesary since one often only needs to switch word-sized groups of data. The flexibility of the fine-grained interconnect increases overhead without increasing usefulness. Thus, to increase the efficiency of RH for general-purpose computation the interconnect should be made less general.

- **Special programming languages:** The lineage of RH from random logic is most strongly reflected in the similarity of the tools used for programming both. Most notably, hardware description languages explicitly describe the activities that should be carried in parallel. In contrast, software languages have a more sequential nature, describing the program as a linear sequence of actions.

  However, if RH is to migrate more toward handling computation, the development environment used to program RH devices must also metamorphose. Currently, an active topic of research is in the determination of how to efficiently bridge the gap between software languages on one side and programming tools and the parallelism available in RH devices on the other side.

- **Long compilation time:** The hardware-design tools used for programming RH devices tend to use a lot of time (hours and even days) in order to achieve high-quality results. This trade-off must be tipped in the opposite way (i.e., lower quality, but fast compilation), to bring compilation time in line with the expectations of software developers.

- **Limited computational resources:** The number of computational elements in an RH device is determined at the time of manufacture. Implementing programs that require more than the available amount of resources is simply not feasible using current technology. In contrast, processors store program instructions in memory, which is cheap and large, and whose capacity grows quickly with new generations. Moreover, practical mechanisms for *virtualizing* the main memory have been used for over three decades. Such mechanisms store the data on much larger disks and use the memory essentially as a large cache for this data. Using virtualization, even programs requiring more memory than physically available can be executed, provided they have good locality. Nanotechnology promises to solve or at least severely ameliorate the problem of limited computational resources: by using molecular devices, several billion computational elements can be assembled in a square centimeter, many more than are currently used by today's most sophisticated circuits.

- **Configuration size and configuration loading time:** As we described previously, the size of the configuration describing the circuit functionality is substantial. Ratios of 100 bits of configuration for describing a 1-bit computation are not unheard of. Large configurations require large storage resources on-chip and, most importantly, require a long time for downloading the configuration from an external medium onto the chip.

  Configuration size and time can be substantially reduced by increasing the computational granularity. For example, when using eight-bit computational units, instead of describing the computation of each one-bit functional unit separately, eight of them share the same configuration.

  The impact of configuration loading time can be alleviated by implementing chips that allow incremental reconfiguration: only as much of the chip should be reconfigured as necessary for implementing the desired functionality. Another practical solution is to completely separate the electrical interfaces for configuration and execution. For such chips, while some part of the chip is being configured, the rest can be used for active computations [GSM+99, GSB+00].

- **Low density and speeds:** The flexibility of RH is offset most seriously by two factors: the reduced number of computational units that can be implemented per unit area, and the slowdown incurred on the electrical signal crossing multiple switching points. A logic gate in a processor or custom hardware device is one order of magnitude more compact than a programmable gate in an RH device. Moreover, substantial space is wasted for the communication and configuration interconnection networks.

  Some CAEN proposals (e.g., [GB01]) solve the density problem by using computational elements and switches which can hold their own configuration, without the need for supplemental memory. The overhead in size of such devices becomes insignificant. The speed overhead, however, is much more difficult to overcome—no effective solutions have yet been proposed.

  Interestingly enough, the low speed of the electrical signals can be construed as an asset of RH devices in the following sense: RH programmers have always had to work with the reality of slow communication links and have had to factor the cost of communication into the balance of the systems. For classical hardware architects, wires were perceived until recently as being essentially free. However, the amazing advances in clock speeds have reached a limit where the propagation delay of wires is no longer insignificant [AHKB00]. Modern designs must accommodate the reality of multicycle latencies for propagating information on the same chip. Classical microarchitectural designs tend to be monolithic, and therefore break down, when the latencies of the communication links become significant. Such architectures will require a major redesign to accommodate multiple unsynchronized clock signals (clock domains). RH devices have therefore always faced this fundamental problem, which classical hardware has only postponed.

- **No ISA:** As described in Section 3.2, a processor is characterized by its instruction set architecture, that is, the set of operations it can carry. The ISA is a contract between the hardware and the software world. The same ISA can be implemented in various ways, but the programs using it continue to work unchanged. This contract is extremely important for decomposing the architecture into clean layers.

  To date no concept equivalent of an ISA has been proposed for RH devices. The lack of an ISA also complicates the task of compiling high-level languages to such architectures: the ISA is a very powerful conceptual handle in the description of programs.

5.7 RH and Fault Tolerance

A subject where the qualities of RH devices differ considerably from either custom hardware or processors is reliability. The “deep” flexibility of the RH

devices enables the construction of computer systems which can reconfigure ”on the fly” to increase reliability or to manage emerging defects. In this section we discuss how the qualities of RH fabrics can be exploited to build robust systems.

5.7.1 Permanent Defect Tolerance

RH devices have a unique quality, which enables boosting the yield (see Section 2.5) almost for free: such chips can be used even when they have a substantial number of defects.

In an RH device practically all programmable gates are equivalent, because each can be configured to implement any functionality. Hence, if some portions of the chip are defective, their functionality can be assumed by other equivalent portions, by simply rerouting the connecting wires and changing the configuration of a few gates (see Fig. 26). Given a *defect map* of a chip, a compiler can use it during the place-and-route process to avoid defective zones and implement perfectly functional circuits using faulty RH. The success of this approach has been demonstrated by the Teramac project [HKSW98] (see below).

5.7.2 Self-Testing

Once the defective parts of a chip are known, they can be circumvented. This can occur, for instance, by using defect-aware place-and-route tools or by shifting parts of the configuration based on the regularity of the chip geometry.

We have not yet explained how defect positions are discovered. For defect discovery we need a testing mechanism which can circumscribe precisely the defective regions. This task is not a simple one because there are few ways of probing a chip.

Testing would be trivial if we could test the individual components. In general, the testing strategy should satisfy the following constraints [MG02]:

- it should not require access to the individual components
- it should scale slowly with the number of defects
- it should scale with fabric size, so that testing does not become a bottleneck in the manufacturing process

The solution is to use the programmability of the chip to implement *self-testing* computations. These are simple computations that can be applied selectively to small parts of the chip. For example, by testing on rows and columns one can precisely assign defects to the intersections of the rows and columns that give wrong results.

RH can be designed to ease the self-testing by using built-in self-testing (BIST). Complex architectures have been devised which can carry complex self-tests [SKCA96, SWHA98, WT97] without an external entity controlling reconfiguration.

An important advantage of BIST is that it allows the testing procedure to be carried out on many parts of the chip simultaneously, reducing the time needed to build the defect map. This is especially important for very large chips, with billions of components, where a sequential testing process would last an inordinate amount of time.

5.7.3 Teramac

How effectively RH devices can provide fault tolerance has been demonstrated by the Teramac project [HKSW98]. Teramac is built RH devices—thousands of which are faulty—and interconnected by a very rich network. Although more than 70% of the RH devices contain some sort of defect, Teramac performs flawlessly.

The key idea behind making Teramac work is that reconfigurability allows one to find the defects and then to avoid them. Before Teramac can be used, it is first configured for self-diagnosis. The result of the diagnosis phase is a map of all the defects. Then, one implements a particular circuit by configuring around the defects.

Place-and-route tools for Teramac have been modified to take into consideration the defect map.

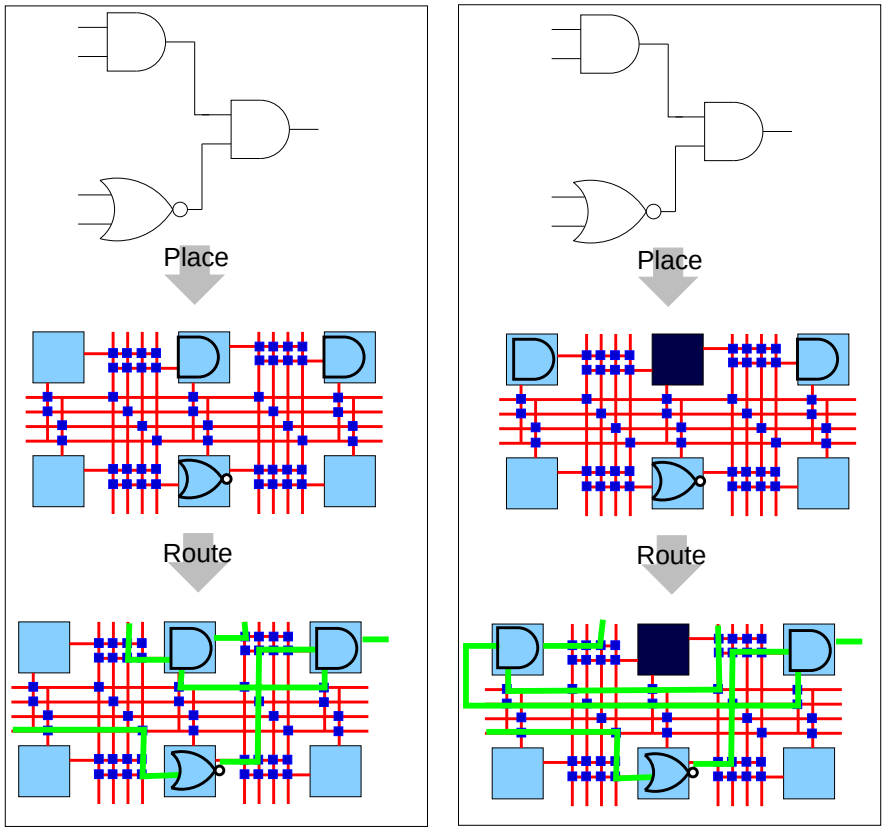


Figure 26: Place and route can avoid chip defects.

For the end-user the presence of defects is practically invisible: self-testing and construction of the defect map, together with automated defect-aware place-and-route, constitute a highly effective fault-tolerance method.

In some sense, Teramac introduces a new manufacturing paradigm, one which trades off complexity at manufacturing time with postfabrication programming. The reduction in manufacturing time complexity makes RH fabrics a particularly attractive architecture for CAEN-based circuits, since directed self-assembly will most easily result in highly regular, homogeneous structures.

5.7.4 Test While Run

Postmanufacturing testing can be extended to allow for continuous testing of the device, even during normal operation. Such devices devote most of the chip area for the main computational task, while a fraction is used for self-testing, in the way described above.

Execution is periodically interrupted and the testing circuitry swaps places with some of the computation; wires are rerouted to connect the displaced circuits correctly, and then computation resumes [ASH+99]. When self-testing discovers faulty areas, these are marked as invalid and never used again.

This procedure allows for an extreme robustness against defects even when they appear during system operation. As long as there is enough space to accommodate the computation and testing, the chip can function even if new defects appear. The cost of this scheme is the area devoted for testing purposes and the time spent for reconfiguration when new chip tiles begin self-testing.

5.7.5 Dynamically Adjustable Fault-Tolerance

The configurability of RH enables it to dynamically change the technique applied for increasing redundancy [SKG00]: for example, when an increasing number of faults is detected, the chip can be reconfigured to carry a higher number of redundant computations. If transient faults are predominant, tests can be carried out using temporal redundancy. When noncritical computations are being carried out, no redundancy need be used at all.

6 Molecular Circuit Elements

We turn now to an examination of the basic devices, wires, and fabrication methods available for constructing molecular electronic computing devices. Our primary focus is on chemically assembled electronic nanotechnology.

CAEN takes advantage of chemical synthesis techniques to construct molecular-sized circuit elements, such as resistors, transistors, and diodes. Chemical synthesis can inexpensively produce enormous quantities (moles) of identical devices, or it can be used to grow devices *in situ*. The fabricated devices are only a few nanometers in size and exploit quantum mechanical properties to control voltage and current levels across the terminals of the device. These molecules are smaller than the physical feature-size limit that can be produced using silicon. Significantly, the operation of these devices requires only tiny amounts of current, which should result in computing devices with low power consumption.

CAEN devices will be small—a single switch for a random access memory (RAM) cell will be approximately 100 square nanometers[17] compared with $100,000$ nm^2 for a single laid-out transistor.[18] To construct a simple gate or memory cell requires several transistors, separate *P*- and *N*-wells, etc., resulting in a density difference of approximately one million between CAEN devices and CMOS. Since very few electrons are required to switch these small devices, they also use less power.

[17] We assume that the nanoscale wires are on 10 nm centers. This is not overly optimistic—single walled nanotubes have diameters of 1.2 nm and silicon nanowires of less than 2 nm have already been constructed.

[18] Even in Silicon-on-insulator where no wells are needed, in a 70 nm process, a transistor with a 4:1 ratio and no wires attached to the source, drain, or gate measures 210 nm x 280 nm. With just a minimal sized wire attached to each, it measures 350 nm x 350 nm.

In this text we largely limit our discussion to molecular devices that have *I-V*[19] characteristics similar to those of their bulk counterparts even when a different mechanism is used to achieve the effect. For example, while the mechanism of rectification is different in a silicon-based *P-N* junction diode from a molecular diode, they both have similar *I-V* curves [AR74]. There are two reasons why we choose to examine systems that can be built from nanoscale devices with bulk-semiconductor analogs: (1) we can apply our experience with standard circuits and (2) we can model the system with standard tools such as SPICE [QPN+02].

Molecular-scale devices alone cannot overcome the constraints that will prove limiting for CMOS unless a suitable, low-cost manufacturing technique is devised to compose them into circuits. Since the devices are created separately from the circuit assembly process, localization and connection of the devices using a lithographic-like process will be very difficult. We must therefore seek other means. Connecting the nanoscale components one at a time is prohibitively expensive due to their small size and the large number of components to be connected. Instead, economic viability requires that circuits be created and connected through self-assembly and self-alignment.

6.1 Devices

While molecular devices of all types have been covered in other sections of this encyclopedia *x & y*, here we examine their effect on the kinds of computing devices that can be built. We examine devices that have direct analogs to bulk devices (e.g., diodes, resistors, transistors, resonant tunneling diodes), as well as devices that have no direct counterpart (e.g., molecular switches). While this is not an exhaustive list of molecular devices, it serves to illustrate the important properties of molecular-scale devices.

[19] Devices are often characterized by the amount of current (*I*) they conduct based on the voltage (*V*) across their terminals. An *I-V* curve is a graph of the current versus the voltage.

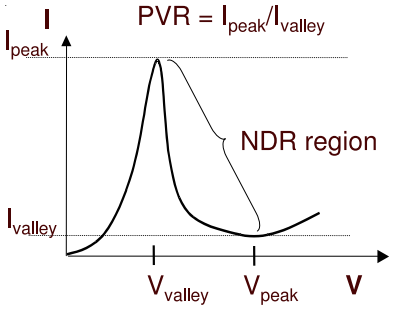


Figure 27: Characteristic curve of a molecular negative differential resistor.

6.1.1 Diodes

A diode is the simplest nonlinear device. It conducts current in only one direction. In 1974, Aviram and Ratner published the first paper on a molecular device. They described a theory which predicted that molecules could behave like diodes. The theory is that molecular diode rectification occurs due to through-bond tunneling between the molecular orbitals of a single D-σ-A molecule. In such a system, electrons flow easily from the donor (D) toward the acceptor (A) through the covalent bond (σ), but not vice versa [AR74]. Since that time, molecules which show rectification have been constructed [Met99]. Additionally, researchers have developed molecular-scale diodes based on the Schottky effect. From a circuit design perspective, the two most important characteristics of these devices are their turn-on voltage (i.e., when they start conducting in the forward direction) and the amount of current that flows in the reverse direction. A perfect diode would start conducting in the forward direction immediately and would conduct no current in the reverse direction. As we shall see later (in Section 8.1), the lower the turn-on voltage, the more useful the device. Unlike silicon-based diodes, molecular diodes have turn-on voltages of less than 0.7 V and forward to reverse ratios of several thousand to one [ZDRJI97]. However, since diodes have no gain, they are not sufficient to build large circuits.

Another molecular device of interest is the resonant tunneling diode (RTD), which is often referred to as a negative differential resistor (NDR) in the molecular device literature. Resonant tunnel-

ing diodes are two-terminal devices that have a region of negative resistance (see Fig. 27). The NDR effect is seen in a variety of molecules, most notably [CRRT99, CR00]. Solid-state RTD-based circuits have a long history [AS62], but were essentially abandoned when transistors became the work horse for circuits. The key feature of an RTD is a region of NDR, where the tunneling current falls as the voltage increases. The ratio of the current at the beginning of the NDR region to the current at the end is called the peak-to-valley ratio (PVR). An additional important characteristic of an RTD is the resistance of the device. Lower resistance leads to faster switching times. While molecular circuits based on RTDs include basic logical operations [EL00] and latches [RG02, NF02], they alone cannot be used to construct large circuits. Although they can be connected in ways that introduce gain into a circuit, without additional devices they do not provide any I/O isolation. In Section 8.3, we discuss their use in building latches and describe how to introduce I/O isolation.

6.1.2 Transistors

As discussed in Section 2.3, transistors are sufficient to construct large circuits. There has been significant recent progress in developing molecular transistors. Transistor-like behavior has been reported in many different kinds of organic and inorganic molecules. In fact, Wind et al. [WAM+02] have reported on a nanotube transistor with twice the current carrying capability of silicon-per-unit width. One advantage of this device is that the gate is localized to the device (i.e., it does not use back-gating).[20] Therefore, more than one device can be present on the substrate. Many other impressive results have been reported on molecular transistors [BHND01, HDC+01a, DMAA01]. All of these devices report gain of more than one, have large current carrying capability, and localized gates. Additionally, some of the reported work has developed both *P*-type and *N*-type transistors, allowing complementary logic to be developed. This last fact is important for building circuits with reduced power requirements. It would therefore seem that these devices would be perfect for building molecular electronic devices. However, assembling them into a circuit is a significant remaining challenge, which we discuss below.

[20] In back-gating the gate that controls the device is the entire substrate on which the device is fabricated. Therefore, when back-gating is used, all the back-gated devices are controlled by a single gate—the substrate.

6.1.3 Switches

Molecular switches, unlike the previous components, are devices that have no single bulk analog. A switch is a device that can be programmed to be either highly resistive (open) or highly conductive (closed). Molecular switches have already been demonstrated [CWB+99, CWR+00]. We focus here on the psuedo-rotaxane developed at UCLA [CWB+99, BCKS94, MMDS97] (see Fig. 28). The psuedo-rotaxane is a two-terminal device which can be viewed as a rod surrounded by a ring. Four voltage points are used to operate this device: program-off, signal-low, signal-high, program-on. The programming voltages cause the device to change state. The signal voltages, which have a lower absolute magnitude than the programming voltages, are used to operate the circuit. Programming is achieved using a high voltage differential applied to the two terminals, causing the ring to move to either the left or the right of the rod, depending on the voltage sign. When the ring is at one end of the rod, the molecule behaves like a wire with relatively low resistance. When the ring is at the other end, the device behaves like an open switch (i.e., it has very high resistance). When in series with a diode, this behaves like a programmable diode: when the ring is at one end the ensemble is an open connection, when on the other end it behaves like a diode. It is, of course, crucial that the signal voltages remain less than the programming voltages. Otherwise, the device will have its state unintentionally changed.

There are two reasons why molecular switches are particularly well suited for reconfigurable computing. First, the state of a switch is stored in the switch itself. Second, the switch can be programmed using the signal wires (i.e., there is no need for additional programming wires). For the rotaxane de-

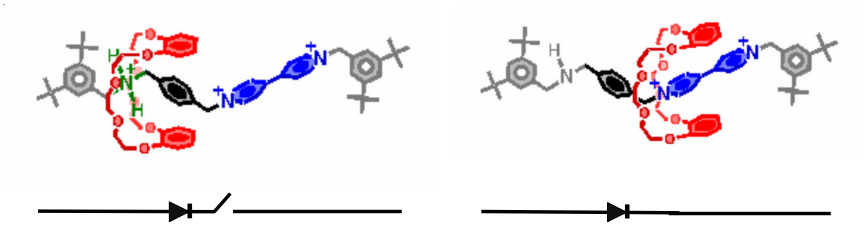


Figure 28: A diagram of a psuedo-rotaxane in series with a diode. When the ring is at the right it has low resistance (i.e., it is a conductor), when at the left it has high resistance (i.e., it is an open circuit). Adapted from Ref. [Sto] with permission J. Stoddart, http://www.chem.ucla.edu/dept/Faculty/Stoddart/research/mv.htm.

scribed above, the state is stored in the spatial location of the ring. Under operating conditions the ring position is stable. On the other hand, a CMOS-based reconfigurable device requires a static RAM cell to control each pass transistor. When the molecular fabric is compared to a CMOS fabric using a 6-transistor static RAM cell to control a pass transistor, we see that a reconfigurable fabric will be at least 10^6 times as dense using molecular electronics. In addition, CMOS reconfigurable circuits require two sets of wires: one for addressing the configuration bit and one for the data signal.

Perhaps a more apt comparison is between molecular switches and floating-gate technology, which also stores the configuration information at the transistor itself. A floating-gate can also be used to create a programmable logic array. Because it requires high voltage for programming, a floating gate transistor is slightly larger than a standard transistor. Its large size makes it, at most, one order of magnitude more dense than the RAM cell, still giving CAEN-based devices a density factor advantage of roughly 10^5. Moreover, floating-gate transistors act as bidirectional devices and therefore are less useful than the molecular switches which can incorporate rectification, acting as diodes.

Another example of a configurable device is a configurable transistor. Fig. 29b shows a schematic of a configurable transistor formed at the intersection of two wires similar in spirit to that reported in [RKJ$^+$00]. Unlike the example shown in Fig. 29a, this device behaves like a transistor when the wires are in physical contact and behaves like two nonintersecting conductors when the wires are detached. The configuration mechanism is not a conformational change in a molecule, but rather a conformational change in the wires. By putting an attractive electrostatic charge on the two wires, they will come together. They will then stick together due to van der Waals forces.[21] They can be separated by putting a repulsive electrostatic charge on the wires.

6.2 Wires

A single device, while interesting, is not useful unless assembled in a circuit. Connecting nanoscale devices together requires nanoscale wires. The primary requirement on a wire is that it quickly transmits (without significantly distorting) a signal from one device to another. Such wires exist and have been made from many different materials, including various metals [PRS$^+$01], silicon [KWC$^+$00], polymers [EL00], and carbon nanotubes [SQM$^+$99]. In addition to the wire, there must be a means for connecting the wire to the device. The connection, or contact, between the wire and the device must also have low signal distortion.

Nanoscale wires can do more than simply conduct, they can also be active devices themselves. For example, carbon nanotubes with diode-like and transistor-like behavior [HDC$^+$01b] and silicon nanowires with transistor-like behavior [CDHL00] have been reported. Another example of wires with active devices are metallic wires which are grown using electrochemical deposition. The fabrication process also allows different materials to be used in

[21] The van der Waals force acts to pull atoms (or molecules) together. It is inversely proportional by the seventh power to distance between the atoms. Clearly this force is only important on the nanometer scale.

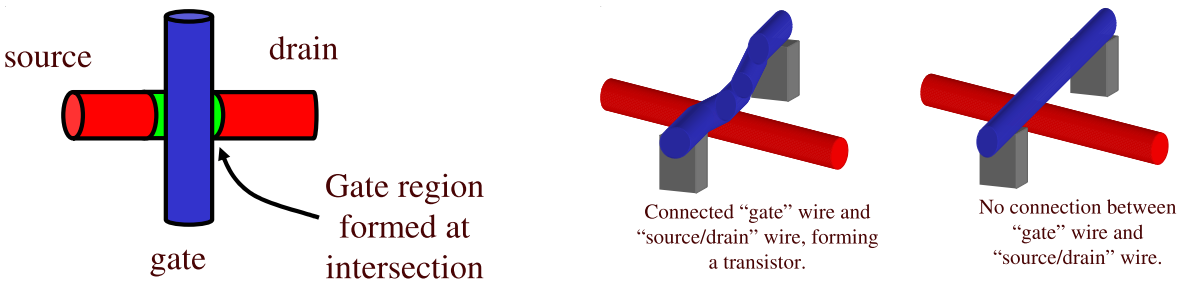


Figure 29: Creating transistors between two wires: (a) fixed and (b) configurable transistors.

the growth of the wire. In addition to making wires with different metals [PRS+01], they can also grow wires with inline molecules [KMM+01, MMK+02, KMM+02]. This technique allows molecules exhibiting rectification, switching, and transistor-like behavior to be incorporated into a wire. In Section 8.3, we show how this technique aids in the construction of a molecular latch. In the end, CAEN blurs the traditional distinction between wires and devices.

Evaluating nanoscale wires is very difficult because the wire/contact/device system is highly intertwined. In fact, some theorize that the nonlinear behavior of molecules is actually a combined effect between the molecule and the contact [Hip01]. However, we can at least characterize the system by the resistance of the wire and the coupling capacitance between neighboring wires and potentially the substrate on which the circuit is built. In the best case, the wire behaves like a one-dimensiional quantum wire (i.e., transports electrons ballistically, having a length-independent resistance). It is theorized that carbon nanotubes behave in this manner.

Their conductance (the inverse of resistance) is expressed in units of e^2/h [Lan57][22] and a perfectly conducting nanotube would have a conductance of $4e^2/h$ or a resistance of approximately 6.5 kΩ independent of its length. This roughly agrees with measurements made for carbon nanotubes by Soh, et. al. [SQM+99], in which they measured resistances for both the contacts and a nanotube as low as 20 kΩ. Using this resistance we can determine a rough lower bound on the RC[23] delay of a molecular computing device; which determines an upper bound on the speed of such a device. Even ignoring the quantum capacitance, the RC delay for 1 micron long wires placed at a 10 nm pitch is .24 ps; at 100 nm pitch it is .13 ps. The relatively long RC constant is due to the high resistance of the nanotube.

Equally as important as the electrical characteristics of the wires and the contacts is the method by which the wire is constructed and the contacts are made. In CMOS-based systems, the wires and connections are made using photolithography. This allows the wires, devices, and connections to be manufactured at the same time, in the same place. This will not be true of molecular systems. In molecular systems the devices will be made separately from the wires and they will then be connected together. The next task is to make the connections economically, yet with high reliability. The ability to achieve this will significantly influence the kinds of structures that can be built using molecular devices.

7 Fabrication

Molecular-scale devices cannot themselves overcome the constraints that will prove limiting for CMOS unless a suitable, low-cost manufacturing technique can be devised for composing them into circuits. For devices created separately from the circuit assembly process, localization and connection of the devices using a lithographic-like process will be very difficult. Other means must be sought. Connecting the nanoscale components one at a time is prohibitively expensive due to their

[22] e is the charge on an electron, h is Plank's constant.

[23] RC is the product of a circuits resistance (R) and its capacitance (C). It indicates the rate at which the circuit can change its value. After RC time units the circuit can charge (or discharge) to approxiamitly 2/3 of its final value.

small size and the large number of components to be connected. Instead, economic viability requires that circuits be created and connected through self-assembly and self-alignment. Several groups have recently demonstrated CAEN devices that are self-assembled, or self-aligned, or both [CWB+99, MMDS97, RKJ+00, AN97]. Advances have also been made in creating wires out of single-wall carbon nanotubes and aligning them on a silicon substrate [TDD+97, PLP+99]. More recently, selective destruction has been used to create desired carbon nanotube structures [ACA01]. In addition, metallic nanowires have been fabricated[24] which can scale down to 5 nm and can include embedded devices or device coatings [MDR+99, MRM+00].

While other sections (see especially *Sections X and Y*) in this encyclopedia detail the many different assembly techniques, here we review some of the techniques in light of their impact on the circuits and architectures that can be created. Mass assembly techniques range from top-down approaches such as photolithography, to bottom-up approaches such as self-assembled monolayers. For reasons stated earlier, we limit ourselves to chemically assembled electronic nanotechnology (i.e., to bottom-up approaches).

7.1 Techniques

Bottom-up, scalable techniques include Langmuir-Blodgett films, flow-based alignment, nanoimprinting, self-assembled monolayers, and catalyzed growth. The common feature among all these techniques, with the possible exception of nanoimprinting,[25] is that they can only form simple structures (i.e., either random or very regular, but not complex aperiodic). An additional characteristic is that the resulting structures usually contain some defects (i.e., the results are not perfect). Finally, it is not possible to predetermine exactly where a particular element will be located in the structure. That is, one cannot deterministically assemble an array of different types of wires into a particular aperiodic pattern.

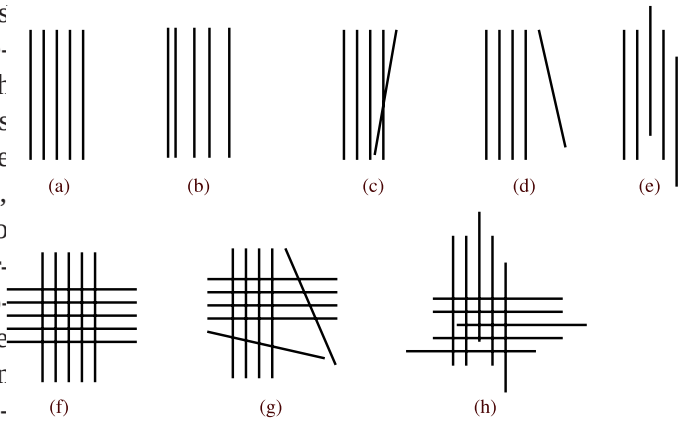


Figure 30: Different defect scenarios using flow-based assembly.

To date, the most complex structures made using scalable bottom-up techniques are two-dimensional meshes of wires.[26] For example, fluidic assembly was used to create a two-dimensional mesh of nanowires [HDW+01]. The nanowires are suspended in a fluid that flows down a channel. The nanowires align with the fluid flow and occasionally stick to the surface they flow over. By varying flow rate and the duration of the flow they were able to vary the average spacing between wires and the density of wires. Further refinements can be made by patterning the underlying substrate with different molecular end-groups to which the nanowires will show preferential binding. By performing the flow operation twice, first in one direction and then in the orthogonal direction, arrays of wires at right angles can be formed.

The above method points out the possibilities and the limitations of bottom-up assembly. On the positive side, one can form one-dimensional arrays and two-dimensional meshes of nanowires (Fig. 30a and f). Negatives to this method include its imprecision and its lack of determinism. Wires will rarely all be

[24] Unlike wires created in traditional processes—i.e., lithography, these wires are "grown" in nanopores.

[25] Nanoimprinting has been used successfully to create irregular structures. However, the process of creating masters combined with direct contact printing may limit the smallest achievable pitch to above 100 nm [XRPW99]. Further, it will be difficult to precisely align the master with the target.

[26] An exception exists for the recent advances in DNA-based assembly (e.g., [WLWS98, Mir00, NPRG+02]). DNA-based assembly has been known to create ordered structures with heterogeneous materials. However, DNA-based methods are even less developed than the methods we explore here.

equidistant from each other (Fig. 30b). Wires that should be parallel may intersect (see Fig. 30c) or be askew (Fig. 30d). The wires may be parallel but may be offset from each other (Fig. 30e). The resulting arrays may contain shorts (see Fig. 30c) or open connections (see Fig. 30g and h). Finally, the technique can never deterministically arrange the wires in the channel. Thus, it cannot produce a complex aperiodic arrangements of the wires.

7.2 Implications

The use of self-assembly as the dominant means of circuit assembly imposes the most severe limitations on nanoscale architectures: it will be difficult to create either precise alignment between components or deterministic aperiodic structures. Chemical self-assembly, as a stochastic process, will produce precise alignment of structures only rarely, and manipulation of single nanoscale structures to construct large-scale circuits is impractical at best. Furthermore, the methods used to assemble nanoscale components are most effective at creating random or, at best, crystal-like structures. These two facts have significant implications on the kinds of circuits and structures that can be realized at the time of fabrication.

The structural implications of bottom-up assembly are:

Connections by overlapping wires: Lack of precise alignment means that end-to-end connections between groups of nanoscale wires will be near impossible to achieve. If all connections between nanoscale wires occur only when the wires are orthogonal and overlap, we reduce the need to precisely align the wires.

Two-terminal devices are preferred: The easiest devices to incorporate into a circuit will be devices with two terminals (e.g., diodes, configurable switches, and molecular RTDs). In the case of wires that do more than conduct (e.g., carbon nanotube diodes), then when two wires intersect, they create an active device between them out of topological necessity.[27] If a molecule is required, then it can be assembled onto one of the wires and then where that wire and another wire intersect an active device will be formed at the intersection. The device will consist of the first wire, the molecule attached to that wire, and the second wire where it touches the molecule on the first wire. Or, in the case of inline devices, one terminal of the molecule is assembled onto the end of the wire as it is grown and then the growth of the wire is resumed, making the other connection to the molecule. In all three cases, there is no need for precise or end-to-end connections because devices are incorporated into the circuit out of topological necessity. A harder problem is using three-terminal devices (e.g., molecular transistors). It will likely prove extremely difficult to attempt the connection of three wires to a molecular transistor *en masse* at the nanometer scale. One possible solution is to arrange for the intersection of the two wires to be the gate of the device. We saw an example of this earlier in Fig. 29.

Active components at cross-points: The most reliable way to make device connections will be to generate them at the intersection of two wires. Such devices can be either two-terminal devices, as described above, three-terminal devices (e.g., [HDC$^+$01b]), or four-terminal devices. In the latter two cases care must be taken to ensure that enough heterogeneity can be introduced into the circuit to utilize the devices. Practical foreseeable nanocomputer designs will therefore have to rely on the placement of active components at the intersections of wires.

Meshes are basic unit: The methods used to assemble wires combined with the method of creating active devices implies that the basic structural unit will be a two-dimensional mesh. More complicated structures will have to be created either by combining meshes together, or

[27] By topological necessity we mean that the outcome is a direct result of the geometric arrangement. In this case, the intersection of two wires creates, simply by being a point of intersection, an active device.

cutting the wires in a mesh to break it up into subarrays.

Nanoscale to microscale connections will have to be sparse: A direct implication of the size difference between the nanoscale and the microscale is that connections between the two will have to be few. If there are many connections between the two worlds, then the density of the nanoscale components will be dictated by the density of the microscale. For example, using a demultiplexer to connect micronscale wires to nanoscale wires would allow at any particular instant 1 of n nanoscale wires to be addressed using $\log_2 n$ micronscale wires. As n grows large, the nanoscale wires dominate the device, not the micronscale wires. We describe several different approaches for interfacing micronscale devices with nanoscale devices in Section 7.3.

The architectural implications of molecular electronics and self-assembly are:

Fine-grained reconfigurable: The most likely assembly processes are best at creating crystal-like structures (e.g., 2D meshes). Therefore, the resulting structures cannot directly implement a complex, aperiodic circuit. To create useful aperiodic circuits will require that the device be configured after it is manufactured.

Defect tolerance: The stochastic process behind molecular self-assembly will inevitably give rise to defects in the manufactured structures. Instead of defect densities in the range of one part per billion (as one gets in silicon), we expect defect densities for CAEN to be as high as a few percent. The architecture will have to be designed to include spares which can be used in place of defective components. Another useful design criterion will be to ensure that when a set of components is assembled into a larger structure (i.e., wires into parallel wires), that the individual components are interchangeable. For example, in a set of parallel wires, it should not matter *a priori* which working wire is used to implement a particular circuit.

Rather than attempting to eliminate all the defects completely with manufacturing techniques, we will also rely on postfabrication steps that allow the chip to work in spite of its defects. A natural method of handling the defects, first used in the Teramac [CAC+97] (see Sections 4.5.2 and 5.7.3), is to design the device to be reconfigurable and then exploit its reconfigurable nature. After fabrication, the device can be configured to test itself. The result of testing could be a map of the device's defects. The defect map can then be used to configure the device to customize the device to implement a particular function.

Locality: As devices and wires scale down to molecular dimensions, the wires become an increasingly important part of the total design. This is true not only of molecular computing but also of end-of-the-roadmap CMOS. Architectures with significant locality (and thus the ability to communicate most frequently over shorter wires) will have an advantage.

7.3 Scale Matching

As we mention above, every nanoscale component cannot be connected to a micronscale component or the density of the overall system would be dominated by the micronscale components. However, all the currently proposed architectures require some connections to the micronscale. The connections are used for a variety of purposes, including connections to CMOS transistors for signal restoration and connections to wires for circuit inputs, power, ground, and the clock for the nanoscale circuits. Additionally, all the reconfigurable architectures require programming signals which must be routed to every reconfigurable switch. For arrays, this requires addressing individual cross-point in the array. Since the programming signals originate in the micronscale world, it would seem that we need to connect every nanoscale wire to a micronscale wire.

Consider the array of wires in Fig. 31. If we had to bring in programming signals on micronscale wires to all the nanoscale wires simultaneously we could not escape the need to connect each of these wires to

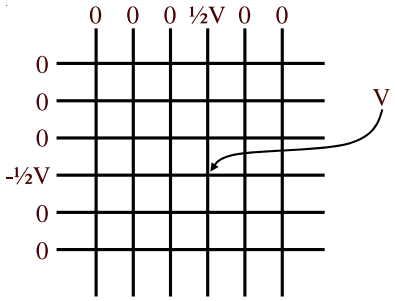


Figure 31: An array of nanoscale wires with programmable molecules at each cross-point. To program a single cross-point, we ensure that the voltage difference between its row wire and its column wire is greater than the programming voltage. All the other wires are kept at zero. The result is that the cross-points that are not being programmed will have voltage drops of less than half the programming voltage.

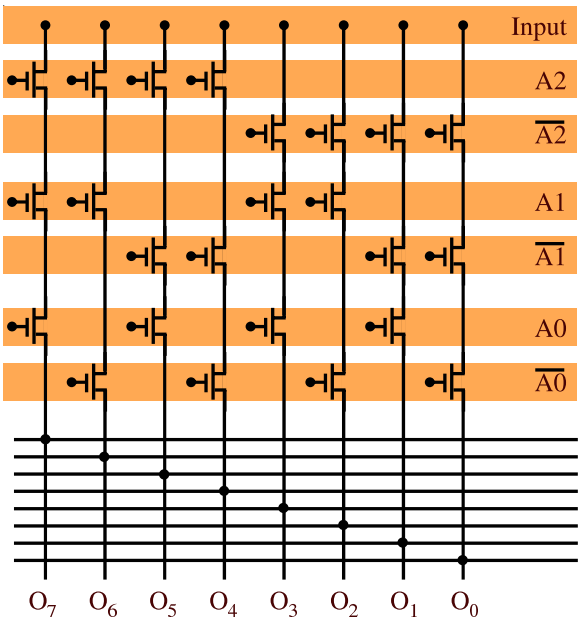


Figure 32: A possible implementation of a 3:8 demultiplexer. The broad lines are micron scale input and address lines. The thin lines are nanoscale lines. These connections between vertical nanoscale lines and horizontal nanoscale lines are not necessary to the operation of the demux.

a micronscale wire. Luckily, we only need to address a single wire in a row and a single column at a time. By raising a single row to half the programming voltage and lowering a single column by half the programming voltage, we have created the proper voltage differential at a single cross-point. By repeating this for each cross-point we can configure the entire array.

Since we only need to raise (or lower) a single wire (in each dimension) at a time, we can use a demultiplexer to select the nanoscale wire. A demultiplexer (demux) connects a single input line to one of n output lines based on an address input. If we encode the address in binary, then we can select one of n outputs with just $\log_2 n$ address lines. If the address lines and the input line are micronscale wires, then we can connect a single micronscale wire to one of n nanoscale wires using only $1 + \log_2 n$ micronscale wires. Since the address lines grow with the log of the output lines, the ratio of nanoscale wires to micron scale wires grows as $\frac{n}{\log_2 n}$. If this ratio is greater than the ratio of the micron to nanoscale pitches, then we will not adversely affect the pitch of the nanoscale wires using a multiplexer. If the nanoscale array has 64 rows and columns, and the pitch is 10 nm, then the array will be 640 nm x 640 nm. This would require two 6:64 demuxs. If the micronscale pitch is 100 nm, then it would be perfectly pitch matched.

Fig. 32 shows a possible implementation of a 1:8 demux. In this implementation each address line is implemented in both its true and complemented form. If the address line is driven high, a logical "1," then the transistors connected to the address line are turned on, the ones on the complemented line will remain off. In this way, one of the eight vertical lines will connect the input line (on the top of the figure) to the bottom of the selected wire. In the figure we have also connected the vertical lines to horizontal nanoscale lines, therefore we are able to connect the horizontal micronscale input line to one of the horizontal nanoscale lines at the bottom of the figure. The area introduced by the demux is $((2\log_2\ n)p_m)(np_n)$, where p_m is the micronscale pitch and p_n is the nanoscale pitch. As the number of nanoscale wires, n, grows large, the overhead for making the connection becomes small.

The demultiplexer above requires precise alignment of the nanoscale and micronscale wires. As we have argued above, such alignment will be difficult achieve. In [WK01], the authors describe a method

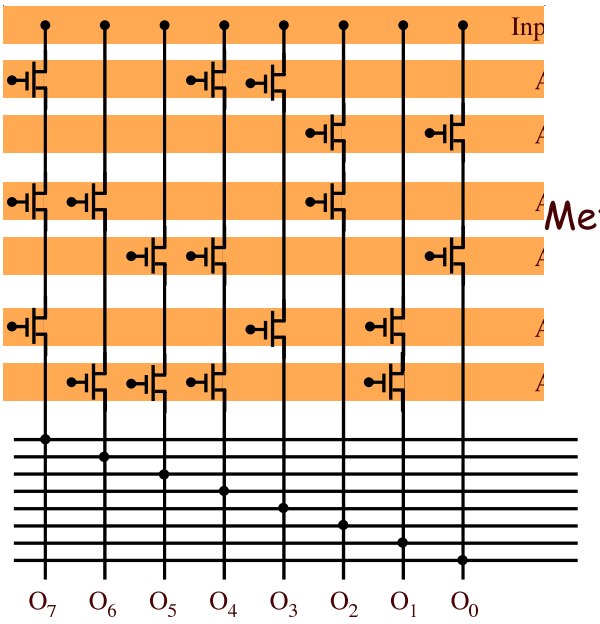


Figure 33: A demux formed using random connections between the microscale wires and the nanoscale wires.

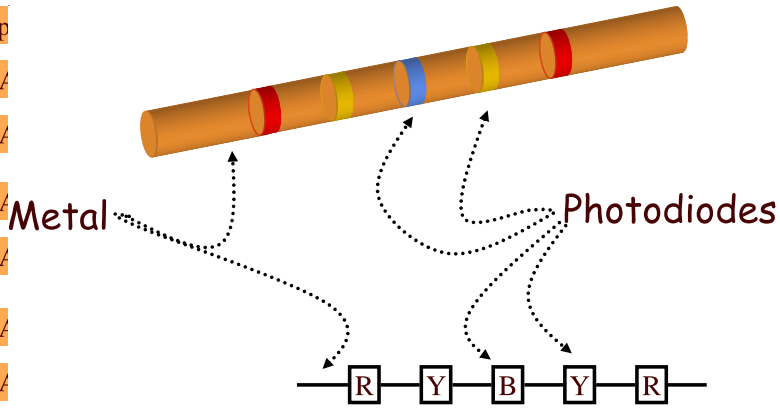


Figure 34: A representation of an optically addressed single-wire demux. This example shows a symmetric single-wire demux that can be programmed with three different frequencies of light.

for constructing a demultiplexer which requires substantially less precision. In fact, as far as layout is concerned, it requires only that the nanoscale wires and the microscale wires are laid out orthogonally. The connections between the wires are made using gold particles that are deposited randomly. The gold particle, when in contact with the nanoscale wire, creates a transistor at the point of contact with the gold particle acting as the gate. Fig. 33 shows schematically what might result for a demux with eight outputs and six address lines. The gold particles are assembled onto the microscale wires resulting in a randomly distributed collection of particles. The microscale wires will act as the address lines in the demultiplexer. A mesh is then created so that the nanoscale wires being addressed overlay the gold particles. Each nanoscale wire will now connect, through the gold particles, to a subset of the microscale wires.

If we can ensure that each nanoscale wire connects to a different set of microscale wires, then the nanoscale wires can be individually addressed by the microscale wires. Of course, we do not know *a priori* which nanoscale wire will be selected by any given address on the microscale wires. We do not even know that only one wire will be selected at a time. This is not as important as it seems as either (1) we may not care which wire is selected as long as only one is selected or (2) we can determine which one is selected postfabrication. As the number of address lines increases, the chances that each nanoscale wire will have a different address increases. For example, $4 \log_2 n$ address lines can address n different nanoscale wires with 50% probability. Furthermore, the mapping of addresses to selected outputs can be determined in $O(\log n)$ time [WK01].

A demultiplexer essentially trades space for time. With direct connections between the nanoscale and the microscale all n nanoscale wires can be connected at once. However, the total area used for such connections scales with n. With a demultiplexer only one connection can be made a time, reducing the bandwidth between the nanoscale and the microscale by a factor of n, but reducing the area required by $\frac{n}{\log_2 n}$.

Another alternative, that trades both more power and more time for space than the demultiplexers above is to use a wire which can have the length of its conducting region configured in a circuit. We call such a wire a single-wire demux (SWD). A SWD is not as general as the demultiplexer but can be used for addressing configuration bits in a nanoscale array. It can be configured to address a subset of the wires to which it is orthogonal. Fig. 34 shows an example of an optically addressable SWD. At the cost of

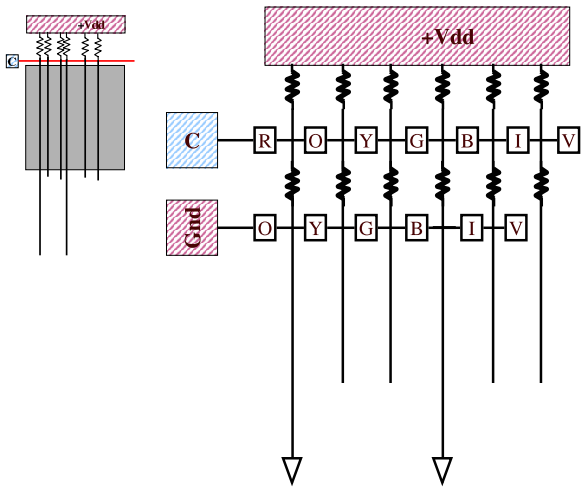


Figure 35: How a SWD might be used to program the individual cross-points of an array.

some power, two SWDs can be used to individually address a single wire out of a set of parallel wires (see Fig. 35). As opposed to the demultiplexer above, this scheme uses more time, more power, and requires light. However, it only needs a constant amount of area. For small sets of wires, it will take considerably less area than the demultiplexer, which scales with the log of the number of wires being addressed.

7.4 Hierarchical Assembly of a Nanocomputer

Bottom-up assembly implies, by definition, a hierarchical approach to assembling complete systems. Having described some of the basic fabrication primitives, we now describe a plausible process for constructing a molecular integrated circuit. The process, a hybrid of molecular self-assembly and lithographically produced aperiodic structure, is inexpensive yet creates structures out of nanoscale components. At the nanoscale, all subprocesses are self-directed. Only at the micronscale do we allow deterministic operations. The process is hierarchical, proceeding from basic components (e.g., wires and switches), through self-assembled arrays of components, to complete systems.

In the first fabrication step, wires of different types are constructed through chemical self-assembly in bulk. Some of these wires may be functionalized, either by coating their exterior or by including in-line devices along the axis of the wire. The next step aligns groups of wires into rafts of parallel wires using flow techniques [HDW+01]. Two rafts can be combined to form a two-dimensional grid using the same flow techniques. The active devices are created wherever two appropriately functionalized wires intersect. By choosing the molecules that coat the wires appropriately, we can make the intersection points ohmic contacts, diodes, configurable switches, or other devices. No precise alignment is required to create the devices, they occur wherever the rafts of wires intersect.

The resulting grids will be on the order of a few microns in size. A separate process will create a silicon-based die using standard lithography. Note, however, that this lithographic process does not have to be "state-of-the-art" since the CMOS components are few and far apart. The circuits on this die will be used to support the nanoscale components. They might include wiring for power, ground, clock lines, interface logic to communicate to the outside world, and support logic for the grids of devices. The previously created grids can then be aligned to the substrate using either techniques such as flow-based techniques or electromagnetic alignment [SNJ+00].

At this point a crystal-like structure has been created composed of simple meshes. In order for this device to perform useful logic it will have to be customized. A combination of reconfigurable switches and wire cutting can be used to customize the simple meshes into complex aperiodic circuit elements. Wire cutting allows a wire to be selectively broken into two pieces [WKH01]. A high voltage is applied at a cross-point which over oxidizes one of the wires. Due to the small diameter of the wire to be cut, the over-oxidation will consume the wire at the cross-point. This results in two disconnected wires.

The resulting device is a reconfigurable fabric, as described in Section 5. Once the device is manufactured, it is tested using techniques described in Section 10. The result of the testing phase is a defect map that is used to avoid the defects in the device. Finally, when the device is used, the desired circuit is loaded onto the device through a configuration. The

end effect of reconfiguration is a customized, aperiodic circuit which can perform a useful function.

The important features of this scenario are that it is based on both fairly random self-assembly at the molecular level (the assembly of the rafts and the wires), and deterministic assembly at the lithographic scale (the placement of the rafts on the CMOS die). It is defect tolerant due to its reconfigurability. It has no end-to-end connections at the nanoscale. All wire connections are made at the intersections of wires.

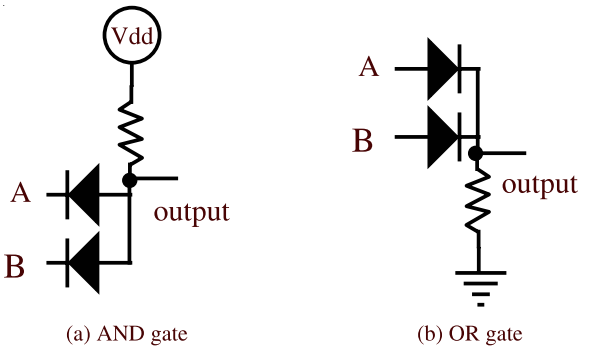


Figure 36: Implementations of an AND and an OR gate in diode-resistor logic.

8 Circuits

The easiest molecular electronic devices to use are those with only two terminals. Such devices can be incorporated into a molecular circuit at the intersection of two wires. However, designing circuits with only two-terminal devices is a challenge. Currently, there are no known two-terminal devices that in and of themselves provide I/O isolation, gain, or even inversion. Therefore, any circuit design methodology will have to create logic functions without these benefits or else find equivalent primitives through collections of the two-terminal devices. If molecular circuits are to scale beyond toy designs, then at some point in the circuit, isolation and gain will have to be introduced.

In this section we reintroduce an old logic design methodology for two-terminal devices: diode-resistor logic. Diode-resistor logic allows for the creation of AND and OR gates. The logic family is made complete (in the sense of Section 2.2) by providing, at the inputs to the circuit, both the inputs and their complements. We then describe a method for introducing gain and I/O isolation using molecular latches based on the NDR molecules described in Section 6.1.

8.1 Diode-Resistor Logic

Without transistors as the active elements, we are reduced to using an older form of circuit design, diode-resistor logic. Diode-resistor logic is a form of threshold logic that relies on the fact that diodes only allow current to flow in one direction.

Fig. 36a shows an AND gate implemented with diode-resistor logic. We call the voltage drop across the diode V_{drop}. If input A is grounded and B is set equal to V_{dd}, current will flow through the diode connected to A and there will be a voltage drop of $(V_{dd} - V_{\text{drop}})$ across the resistor. Note that there will be no current flow through the diode connected to input B, since it is back-biased. If we set a logic "0" to be any voltage between ground and several times V_{drop}, then the output will be a logic "0." Similarly, if B is low and A is high, then current will flow through B's diode and not A's. The output will still be "0." If both inputs are low, then the output will also be low. Only if both A and B are high will the output be high. In that case neither diode conducts and there is no voltage drop across the resistor.

Fig. 36b shows an OR gate. In this case, the resistor is a pull-down resistor connected to ground. The only way to make the output low is if both A and B are low. If either A or B is high, then current will flow through a diode. This will cause the output to be $V_{dd} - V_{\text{drop}}$.[28] If both A and B are low, then the diodes will both be back-biased, no current will flow, and the output will be low.

[28] Both the AND and OR circuits shown here illustrate the need to develop diodes with a very small V_{drop}. The smaller V_{drop} is the closer the output high output will be to a logic "1" for the OR gate and the closer a low output will be to a logic "0" for an AND gate.

There are several problems with this technology. First, the voltage drops across the diodes tend to reduce the separation between a logic one and a logic zero. For example, suppose $V_{dd} = 5V_{\text{drop}}$ and a logic zero is any voltage less than $2V_{\text{drop}}$ and a logic one is any voltage above $3V_{\text{drop}}$. Suppose there are four AND gates connected in series, with the output of each gate connected to both A and B of the next gate. Then, if the input to the first gate is 0 V, then the output from the final gate will be $4V_{\text{drop}}$, which is interpreted as a logic one when it should be a logic zero. Increasing V_{dd} or decreasing V_{drop} helps but does not solve the basic problem (i.e., that diode-resistor logic does not implement a restoring logic). A further problem is that there is no I/O isolation. We address both of these issues in the next section.

The third problem with diode-resister logic is that it is not a complete logic family. That is, one cannot invert a signal using only diode-resister logic. This flaw is not fatal, in that if the inputs to a logic function include both the signals and their complements, then one can compute both the outputs and their complements. This result is based on de Morgan's law, which states that $A \vee B = \overline{\overline{A} \wedge \overline{B}}$ and $A \wedge B = \overline{\overline{A} \vee \overline{B}}$. Therefore, we only have to ensure that all inputs to the device from the outside world include their complements. We arrange this by inverting outside signals with CMOS as part of the external interface. Then whenever a logic function computes f, we also compute $\overline{f}$.

Finally, this logic family consumes power even when the logic elements are not switching. The static power dissipation is V^2/R. To make this a viable technology the resistors will have to be very large, on the order of 100 MΩ. This will certainly affect the speed of the circuit, since the RC time constant is directly proportional to the resistance in the circuit. Fortunately, the capacitances of molecular scale wires are small, on the order of a couple of tens of attofarads, due to their small geometries.

8.2 RTD-Based Logic

Resonant tunneling diodes (RTDs) can form the basis for a logic family. RTD-based logic is also based around the nonlinear characteristics of a diode. However, instead of a simple rectifying device, it is based on a diode with a region of negative differential resistance, as shown in Fig. 27 in Section 6.1. This logic family is based on bulk semiconductor tunnel diodes from the early 1960s (e.g., [AS62, Car63]).

Most of the work employing tunnel diodes uses them in conjunction with transistors [MSS+99, GPKR97, PG97]. Such circuits are not practical at the molecular level because they require complicated arrangements of transistors and tunnel diodes. If the transistors themselves were freely available, standard logic families would be sufficient for implementing circuits.

Work by Nackashi and Franzon [NF02] investigates using only NDRs[29] and other two-terminal devices as the basis for a logic family. The nonlinearity of the NDR is used in conjunction with a network of resistors to create NAND and NOR gates, creating a complete logic family. They found that circuits based on such devices are extremely sensitive to small variations in the NDR behavior (particularly the voltage at which the peak current flows) and the values of the resistors. Furthermore, such logic families do not provide gain or I/O isolation.

8.3 Molecular Latches

Neither diode-resistor logic nor RTD-based logic is scalable for the reasons stated above. Therefore, if either family is going to be used to compute logic functions, there must be another kind of device that can introduce gain and provide I/O isolation. While it might be possible to use either molecular or CMOS transistors, here we explore a more fabrication-friendly alternative based only on two-terminal devices. The latch we describe is motivated by early work on tunnel diodes [AS62] and more recently on compound semiconductor RTDs [MSS+99]. The molecular latch provides the important properties of signal restoration, I/O isolation, and noise immunity.

[29] An NDR is distinguihed from an RTD only in manufacture and we will use these two acronyms interchangably.

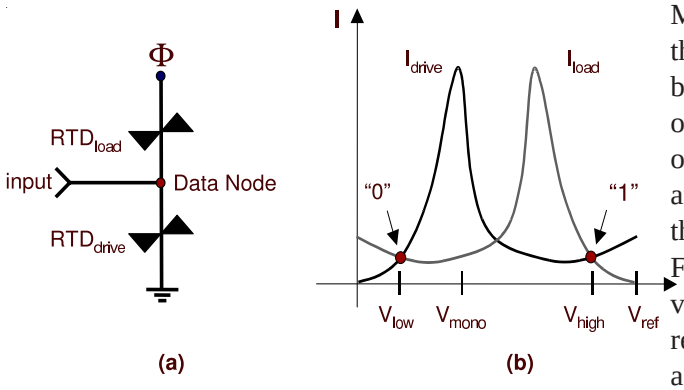


Figure 37: Load lines for the molecular latch. During a latching operation the voltage at Φ is lowered to V_{mono} and then raised back to V_{ref}.

8.3.1 Molecular Latch Operation

The molecular latch is constructed from a pair of molecular RTDs (also called NDRs), described in Section 6. Fig. 27 shows an *I*-*V* curve of a representative RTD. The key feature of an RTD is a region of negative differential resistance, where the tunneling current falls as the voltage increases. The ratio of the current at the beginning of the NDR region to the current at the end is called the peak-to-valley ratio (PVR). Molecular RTDs that operate at room temperature with PVRs of 10 or more have already been realized [CRRT99, CR00].

Fig. 37 shows the arrangement of the RTDs in a molecular latch and a load-line diagram for the molecular latch at a bias voltage V_{ref}. The state of the latch is determined by the voltage at the node between the two RTDs. The load line diagram shows two stable states, V_{low} and V_{high}, and a third metastable state. Small voltage fluctuations in the metastable state will push the circuit into one or the other of the stable states. The low state represents binary "0" and the high state binary "1."

The state of the latch is changed by temporarily disrupting and restoring the equilibrium to one of the two bistable states. The bias voltage is temporarily lowered to V_{mono}, causing a shift of the load line to the left such that the circuit has only one stable state. The bias voltage is then returned to V_{ref}. During the evolution from the monostable to bistable state (the monostable–bistable transition, or MBT), current introduced to the data node will flow through the drive RTD. If the total charge introduced by this current during the transition exceeds a threshold value, the circuit will switch to the high state, otherwise the circuit will switch to the low state. The amount of current necessary to force the latch into the high state is determined by the PVR of the RTD. For a well-matched RTD pair, this current can be very small [AS62]. A higher PVR requires more current to set the high state but provides greater stability against current variation. For the circuit to function correctly, the drive RTD must be more resistive than the load RTD (i.e., R_{ser} in Fig. 38a must be larger for $\text{RTD}_{\text{drive}}$).

RTD latch technology has already been exploited in constructing a GaAsFET logic family by Mathews et al. [MSS+99]. They describe a series of logic gates constructed from RTD latches, FETs, and saturated resistors that display high switching speeds and low power dissipation. In CAEN, the RTD latches are used for a different purpose: they provide voltage buffering by storing the state of previous computation for use in later computation, and signal restoration to either the "0" or "1" voltage with each latch. All computation is performed by diode-resistor logic and the latch restores the signal for a later computational stage. Work in the 1960s with tunnel diodes and threshold logic was hampered by high manufacturing variability [HB67]. The molecular RTDs discovered so far have higher PVRs than semiconductor-based devices, which, as we explain below, may alleviate some of these problems.

8.3.2 Molecular Latch Simulations: Restoration and Reliability

Since molecular-scale RTD latches have yet to be realized, we conducted simulations of their behavior using SPICE. We used a similar device model as Mathews et al. for each RTD [MSS+99]. The device model is depicted in Fig. 38a. The RTD is modeled as a series resistance R_{ser}, with a parallel capacitance C_{RTD} and voltage-dependent current source $I(V)$. This meshes well with the proposed construction of molecular-scale RTDs, which is accomplished by connecting two segments of molecular-

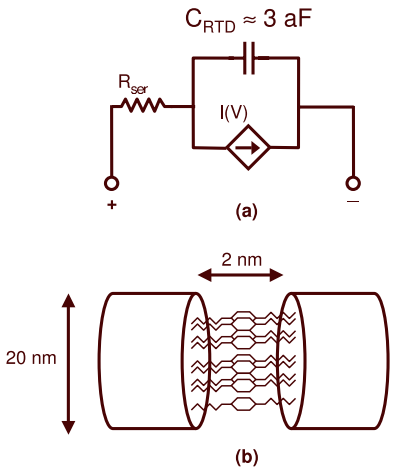


Figure 38: (a) Device model for RTD. (b) Possible construction technique for molecular RTD as an inline device (not to scale).

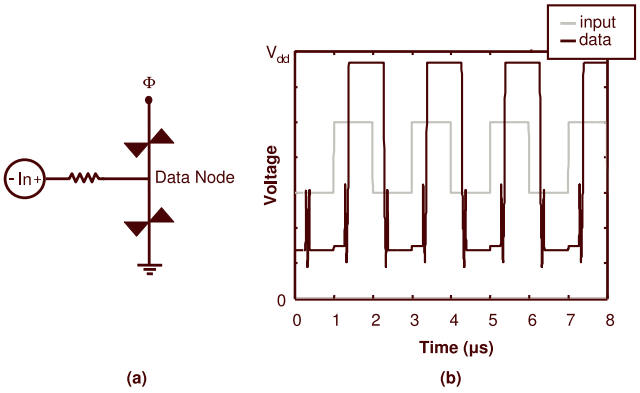


Figure 39: (a) Simple molecular latch circuit used in testing. (b) Results of single stability experiment. Note voltage "gain" and memory effect.

scale metal wire with a number of identical RTD molecules (see Fig. 38b). Based on the estimated length and dielectric of the molecules and diameter of the molecular wires, we estimate a capacitance of 2.7 aF for each RTD. We constructed model equations for the voltage-dependent current source to mimic the *I-V* curves reported by Reed and Tour at 190 K [CWR$^+$00]. To explore the RTD parameter space, we also constructed models for several different "ideal" molecular RTDs in which the V_{peak} and V_{valley} locations and PVR were varied.

Fig. 39b shows the results of a simple RTD simulation using an idealized molecule as the basis of the RTD latch circuit shown in Fig. 39a. The clock signal is shaped to provide a MBT in every cycle and to maintain the bistable state for the remainder of the cycle. Note that the cycle time was deliberately made longer than required to ensure equilibrium before the next cycle. The figure shows that the voltage is restored to the levels predicted by the RTD load line and the choice of V_{ref}. In contrast to the "ideal" molecule, the relatively high V_{peak} of the Reed-Tour molecule would produce narrowly spaced "0" and "1" voltages for any reasonable choice of V_{ref}significantly reducing the noise margins of any circuit using this kind of latch. The molecular RTD latch becomes more practical as V_{peak} becomes lower. The closer V_{peak} is to 0 V, the larger the separation of low and high signals that can be produced at a realistically achievable V_{ref}.

To determine the envelope of stability for the RTD, we simulated the behavior of this simple circuit while varying the "low" and "high" input voltage, the input current, and the relative sizes of the load and drive RTDs. In simulation, simple RTD latches in isolation appear to be stable over the range of variability likely to be encountered in their synthesis. Stability has proven a significant stumbling block to the application of semiconductor RTDs, as device variability often falls outside the required range for incorporation into circuits [HB67]. Molecular RTDs have an advantage in that all molecules incorporated into the RTD are identical. In addition, molecular RTDs have far bigger PVRs than conventional RTDs, which improves the stability of latches constructed with them [MSS$^+$99].

8.3.3 Combinations of Latches: I/O Isolation

In molecular circuits, latches are used to buffer and condition the output of a diode-resistor combinational circuit so that it can serve as input to the next one. We therefore need to consider the interactions of multiple latches. The simplest circuit using more than one latch is a delay circuit, as shown in Fig. 40. Because the latch data node emits the correct voltage only when the latch clock is in the high state, a two-phase clocking scheme (as shown in Fig. 41) is required to propagate the signal through the circuit.

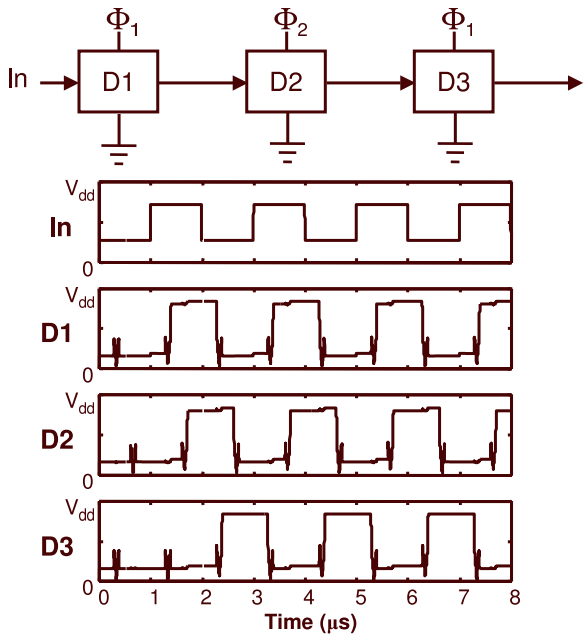


Figure 40: Simple 3-stage delay circuit using molecular latch from Fig. 42.

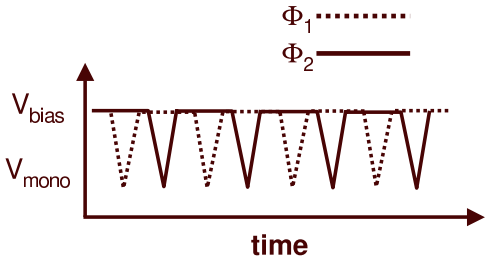


Figure 41: Two-phase clocking scheme used for series latches.

Latches connected in series have alternating clock phases.

It is clear that the lack of I/O isolation provided by transistors is problematic for a latch consisting solely of an RTD pair. Without intervening devices, all data nodes must exist at the same voltage, and even with additional linear circuit elements current is free to flow backward to set upstream latches. To provide the necessary isolation characteristics, we incorporate several additional devices into the latch, as shown in Fig. 42.

A molecular-scale diode is added to prevent current from a downstream latch from flowing to set an upstream latch. Without the diode, setting a downstream latch will prevent the immediately upstream latch from resetting properly. Molecular diodes with very low voltage drops and reverse current flow have already been realized [KMM+02]. The resistor, R, immediately following the latch data forces enough current into $\mathrm{RTD}_{\mathrm{drive}}$ to allow it to switch high, while allowing enough current into the next latch to determine its value. Without the resistor, latches in series would act as a current divider, and insufficient current would flow through each $\mathrm{RTD}_{\mathrm{drive}}$ to set it to the "1" state.

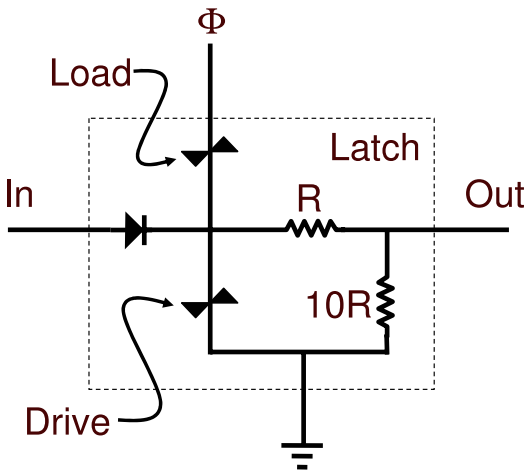


Figure 42: The complete molecular latch.

To understand the rationale for the resistor to ground, consider the delay latch shown in Fig. 40 and the circuit state (In=Low, D1=Low, D2=High, D3=High) at the instant before the Φ_2 clock cycle. As the Φ_2 clock voltage is lowered to V_{mono}, the diodes in both D2 and D3 are reverse biased, preventing any current discharge from the capacitance of latch D2. Since discharge is necessary for the latch to enter the low state, we provide a path to ground through a large resistor. The value of this resistor must be large to prevent current loss that would cause latch-setting failure. However, the resistor should also be as small as possible in order to minimize the latch reset time.

We simulated several simple circuits using the above latch configuration, including the delay circuit in Fig. 40 and the AND, OR, and XOR circuits shown in Fig. 43. The circuits calculate the correct values, taking into account the delay required to tra-

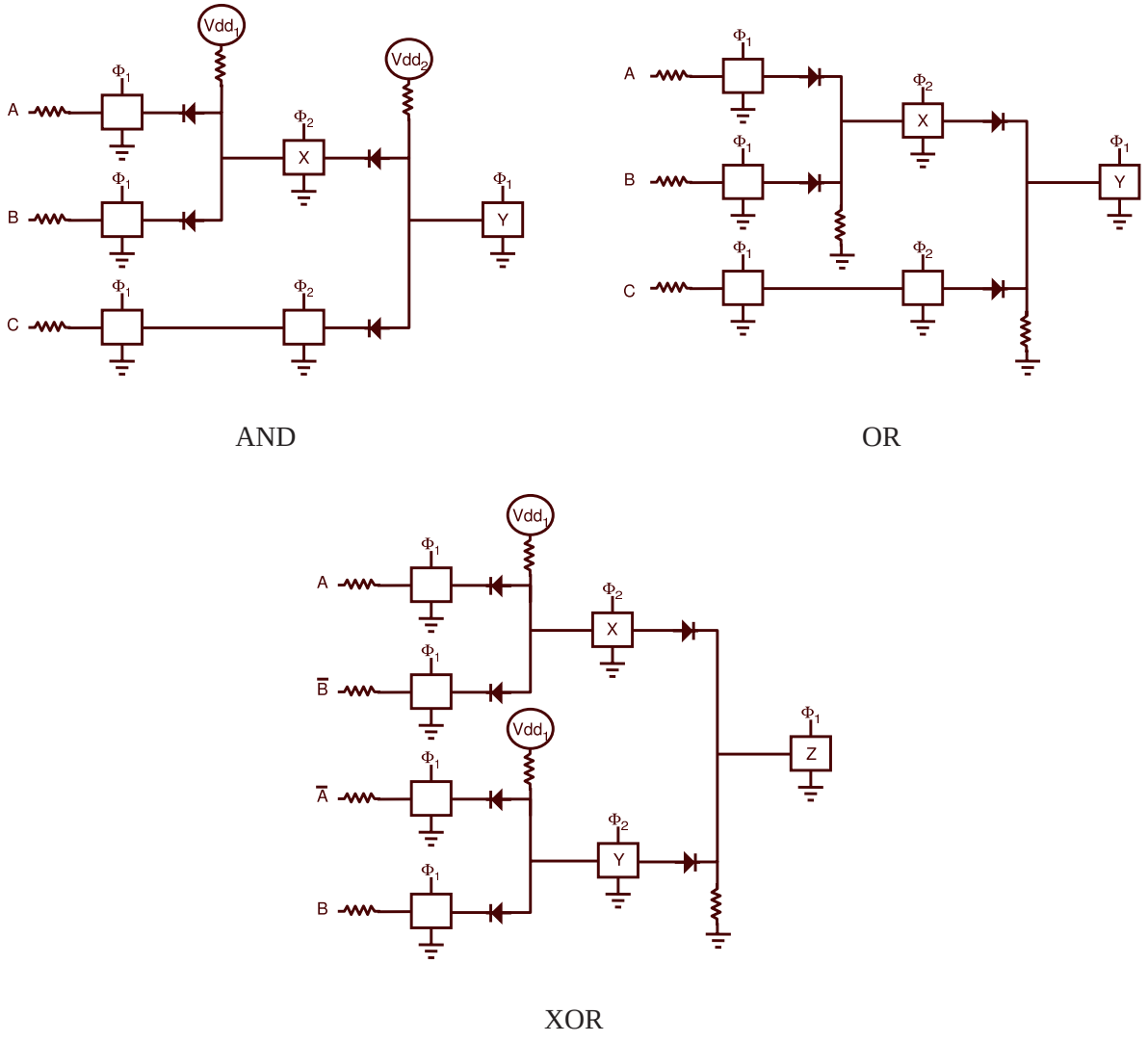


Figure 43: AND, OR, XOR circuits using diode-resistor logic and intermediate molecular latches.

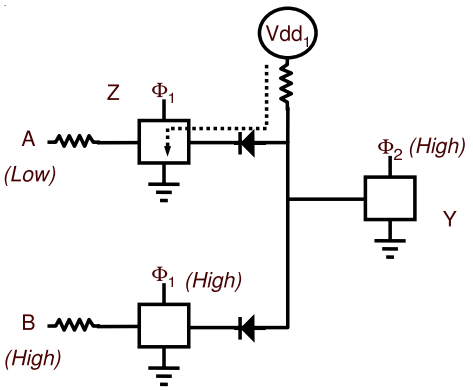


Figure 44: Lack of I/O isolation in AND circuit provides path from V_{dd1} to ground.

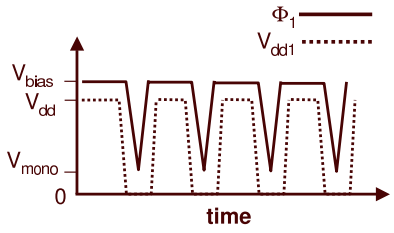


Figure 45: Clocking scheme used for V_{dd}.

verse intervening latches. Circuits with more logic levels were also simulated successfully.

One additional complication resulting from a lack of I/O isolation was discovered for the AND circuit. Consider the circuit state (A=low, B=high, Y=high, Z=high) immediately before Φ_1 begins its cycle (Fig. 44). The diode of Z is reverse biased, so the current in V_{dd1} will flow through the AND diodes into the latches and to ground, preventing them from being reset. The latch cannot be protected by a diode to prevent this, because doing so leaves no path to sink current from the pull-up voltage. Without a path to ground, the pull-up would cause Z to be set high incorrectly. To counteract this, we introduce a clocking scheme for the pull-up voltage as well, as shown in Fig. 45. The pull-up voltage is temporarily brought to ground during the MBT of the preceding latch. This removes the forward influence and allows the latches to be set properly. Unfortunately, in a real circuit this will likely have deleterious consequences for the RC constant.

8.3.4 Latch Summary

In simulation, the molecular latch is capable of restoring voltage levels to the two stable values and of providing current input, as determined by the *I-V* curve of the underlying RTD. It appears to be stable against variability brought about by manufacturing, as long as the drive RTD can reliably be made more resistive than the load RTD.[30] Because of the relatively low current required to switch states, one molecular latch can drive several other latches (i.e., latches can have moderate fan-out). In combination with the clocking scheme described above, the latch also provides the necessary I/O isolation to ensure proper calculation.

Some caveats are necessary, however. Because the RTD devices discovered so far are extremely resistive, the RC constant for circuits incorporating them is very long (on the order of hundreds of nanoseconds). Clearly, this will need improvement before these devices become practical. Also, the relative sizes of the various resistors must be carefully controlled during manufacture to ensure that they are properly matched. This will allow more devices to attain proper I/O isolation without adversely affecting the time constants for the circuit. (Note, however, that controlling the resistance promises to be easy compared to aperiodic placement of devices.) Lastly, the use of a clocked pull-up voltage may result in additional complications for the design of the CMOS support circuitry. However, the similarity in waveforms between the latch clock and V_{dd} clock may alleviate some of these problems.

8.4 Molecular Circuits

CMOS designers have long enjoyed the benefits of using a physical device, the transistor, which has been close to ideal. Furthermore, they have had the freedom to connect devices together in whatever topology they chose. With molecular electronics (and potentially with end-of-the-roadmap CMOS), this will not be true. Instead of a single device

[30] The actual RTDs need not be manufactured differently. Instead the resistance at the connection to drive RTD can be increased; which can be accomplished quite easily.

providing all the required features (switching logic, isolation, restoration of signals, and gain), different devices will have to be used each of which potentially only provides a single one of these features. In this section we have proposed a combination of diode-resistor logic (for logic), a molecular latch (for restoration of signals and memory), and a combination of diodes, resistors, and a clocking methodology for isolation. There are other alternatives to this combination. For example, the bulk of the logic could use arrays of diode-resistor logic. State could be stored using molecular latches. Another alternative is to intersperse occasional transistors in the circuit for I/O isolation and gain. Whatever the solution, with high likelihood, circuit designers using molecular electronics will have to obtain the behavior they want with a combination of devices. For example, one might ask, if transistors are available for isolation and gain why not use only transistors. The reason is that transistors are likely to be hard to incorporate in the circuit for reasons described earlier. So, limiting them will increase the overall density and reduce the overall cost of the final circuit.

Circuit designers currently use an abstraction of the transistor as the basic building block of logic. The abstraction combines three essential components: a logical switch, isolation of inputs from outputs, and, signal restoration. To build memory devices several transistors need to be combined. Examining molecular devices suggests that breaking the abstraction for an ideal transistor into its primitive components may provide intellectual leverage for constructing molecular circuits. We propose the *SirM* abstraction for molecular components. SirM stands for switch, isolator, restorer, and memory. The SirM device abstraction allows circuit designers to construct circuits from devices each of which may only display one of these four characteristics. Using the previous example, we might construct our logic from combinations of diodes and resistors (the switch component), molecular latches (the restorer component and the memory component), and a clocking methodology that provides isolation. Alternatively, the switch component might come from diodes and resistors, the isolator and restorer components from molecular transistors, and the memory component from latches.

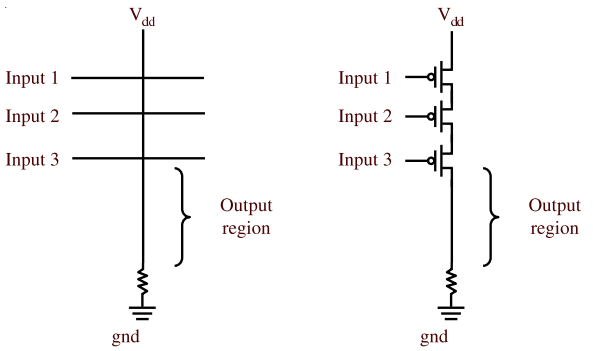


Figure 46: Example of a non-configurable *P*-type nanowire FET. The diagram on the left shows the crossing wires and on the right it shows the circuit formed.

SirM recognizes that one device does not have to provide all the features necessary for a composable scalable logic family. By identifying the primitives necessary (switch, isolator, restorer, and memory) circuit designers and device designers are given the freedom to optimize the types and combinations of devices in ways that they have not been able to before. Now, one can talk about complete device families, instead of single devices, that provide all the necessary components for a complete logic family. A complete device family is one that provides all components of SirM and therefore can be used to construct a scalable complete logic family.

8.5 Molecular Transistor Circuits

Nanowire FETs have been demonstrated which have the potential to form the isolator/restorer components of SirM or potentially to be the core of a complete logic family. Unlike CMOS, this will not be a complementary logic family so, like diode-resistor logic, it will burn power even when it is not switching. Fig. 46 shows a nanowire *P*-FET NOR gate made only with crossing wires. This device, if configurable, could be put into an array so that the inputs all run horizontally. Each vertical wire could be configured to compute a different function on the inputs. DeHon proposes an architecture (see Section 9.3) based on this configurable device [DeH02]. If the device is not configurable, then it can be used as the isolator or restorer component of a complete device family.

9 Molecular Architectures

There are three basic approaches to molecular architectures: random, quasi-regular, and deterministic. The random approach assumes only the most basic manufacturing primitives and only requires that molecules and wires be assembled in random patterns. On the other end of the spectrum, the deterministic approach assumes that complex, aperiodic structures can be precisely built. It requires precision similar to that used in lithographically based semiconductor manufacturing. The quasi-regular approach takes a middle road. It requires that two-dimensional cross-bars be manufacturable. It does not require that the wires in the cross-bar be laid out in any particular order.

We begin this section with a detailed description of the nanoFabric architecture, which is an example of a quasi-regular molecular architecture. We then review other approaches in all three categories.

9.1 NanoFabrics

The nanoFabric is an example of a quasi-regular architecture designed to overcome the constraints associated with directed assembly of nanometer-scale components and exploit the advantages of molecular electronics. Because the fabrication process is fundamentally nondeterministic and prone to introducing defects, the nanoFabric must be reconfigurable and amenable to self-testing. This allows us to discover the characteristics of each nanoFabric and then create circuits that avoid the defects and use only the working resources. For the foreseeable future, the fabrication processes will only produce simple, regular geometries. Therefore, the proposed nanoFabric is built out of simple, two-dimensional, homogeneous structures. Rather than fabricating complex circuits, we use the reconfigurability of the fabric to implement arbitrary functions postfabrication. The construction process is also parallel. Heterogeneity is introduced only at a lithographic scale. The nanoFabric can be configured (and reconfigured) to implement any circuit, like today's FPGAs. However, the nanoFabric has several orders of magnitude more resources.

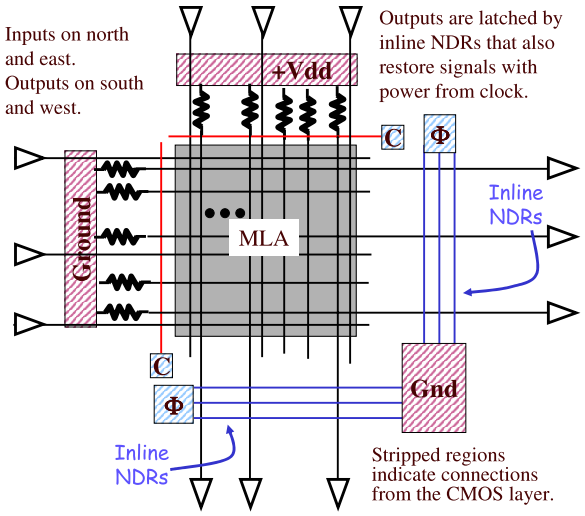


Figure 47: Schematic of a nanoBlock.

The nanoFabric is a planar mesh of interconnected nanoBlocks. The nanoBlocks are logic blocks that can be programmed to implement a three-bit input to three-bit output Boolean function and its complement (see Fig. 47). NanoBlocks can also be used as switches to route signals. The nanoBlocks are organized into clusters (see Fig. 48). Within a cluster, the nanoBlocks are connected to their nearest four neighbors. Long wires, which may span many clusters (long-lines), are used to route signals between clusters. The nanoBlocks on the perimeter of the cluster are connected to the long-lines. This arrangement is similar to commercial FPGAs (allowing us to leverage current FPGA tools) and has been shown to be flexible enough to implement any circuit on the underlying fabric.

The nanoBlock design is dictated by fabrication constraints. Each side of the block can have inputs or outputs, but not both. Therefore, the I/O arrangement in Fig. 48 is required. We have arranged it so that all nanoscale wire-to-wire connections are made between two orthogonal wires so that precise end-to-end alignment is not needed. Figures 49b and c show how the outputs of one nanoBlock connect to the inputs of another. We call the area where the input and output wires overlap a *switch block*. Notice that the outputs of the blocks are either facing south and east (SE) or north and west (NW). By arranging the blocks such that all the SE blocks run in one

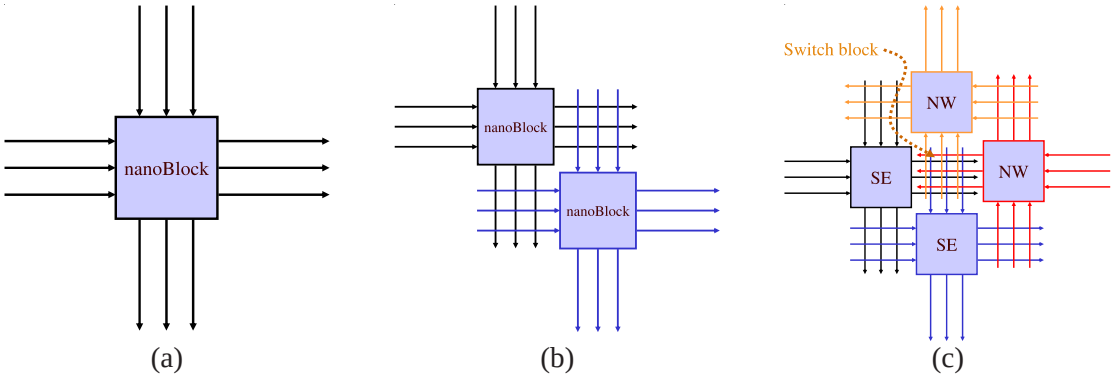


Figure 49: Diagrams showing how the inputs and outputs of a nanoBlock connect to their neighbors.

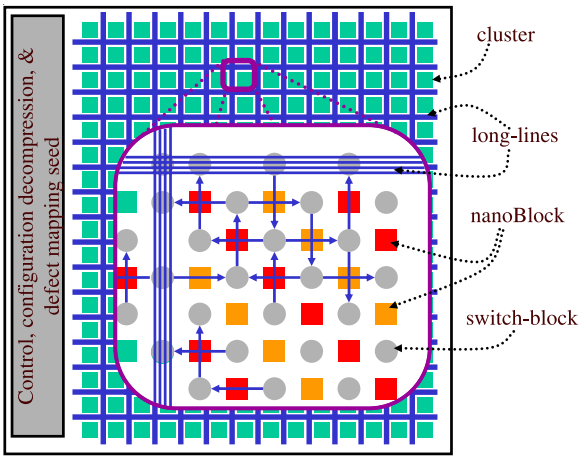


Figure 48: The layout of the nanoFabric with a partial blowup of a single cluster and some of the adjacent long-lines.

diagonal and the NW run in the adjacent diagonal, we can map any circuit netlist onto the nanoFabric. Since the nanoBlocks themselves are larger than the minimum lithographic dimension, they can be positioned precisely at manufacturing time in the desired patterns using lithography.

In addition to the intracluster routing there are long-lines that run between the clusters to provide low-latency communication over longer distances. The nanowires in these tracks will be of varying lengths (e.g., one, two, four, and eight clusters long), allowing a signal to traverse one or more clusters without going through any switches. This layout is essentially that of an island-style FPGA [SBV92]. This general layout has been shown to be efficient and amenable to place-and-route tools [BR97]. Notice that all communication between nanoBlocks occurs at the nanoscale. The fact that we never need to go between nanoscale and CMOS components and back again increases the density of the nanoFabric and lowers its power requirements.

The arrangement of the clusters and the long-lines promotes scalability in several ways. First, as the number of components increases we can increase the number of long-lines that run between the clusters. This supports routability of netlists. Second, each cluster is designed to be configured in parallel, allowing configuration times to remain reasonable even for very large fabrics. The power requirements remain low because we use molecular devices for all aspects of circuit operation. Finally, because we assemble the nanoFabric hierarchically we can exploit the parallel nature of chemical assembly.

9.1.1 NanoBlock

The nanoBlock is the fundamental unit of the nanoFabric. It is composed of three sections (see Fig. 47): (1) the molecular logic array, where the functionality of the block is located, (2) the latches, used for signal restoration and signal latching for sequential circuit implementation, and (3) the I/O area, used to connect the nanoBlock to its neighbors through the switch block.

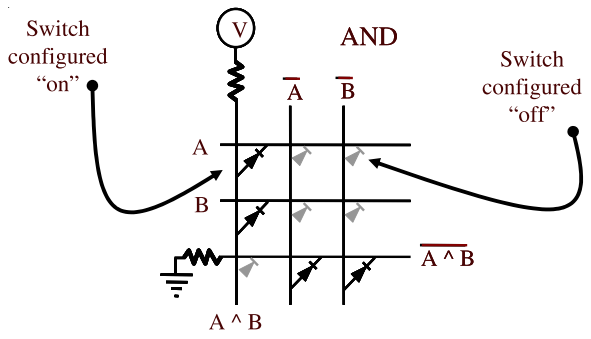


Figure 50: An AND gate implemented in the MLA of a nanoBlock.

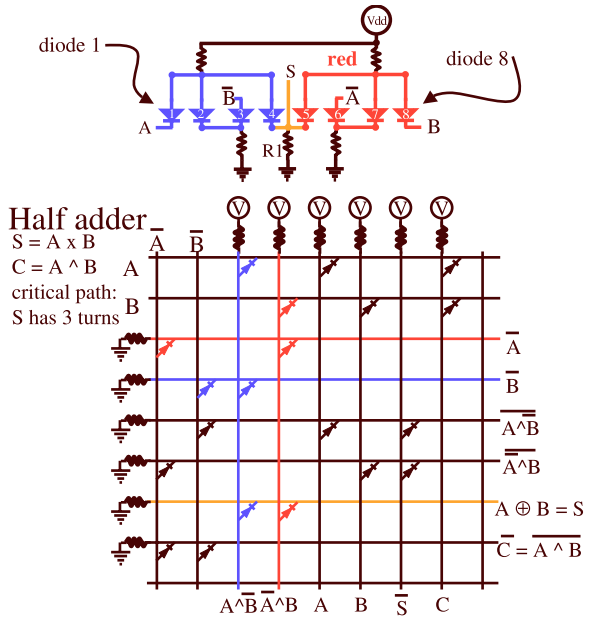


Figure 51: A half-adder implemented in the MLA of a nanoBlock and the equivalent circuit diagram for the computation of A XOR B = S. Only S, $\overline{S}$, C, and, $\overline{C}$ are outputs of this circuit. The rest of the lines are intermediate results and label only for reference.

The molecular logic array (MLA) portion of a nanoBlock is composed of two orthogonal sets of wires. At each intersection of two wires lies a configurable molecular switch. The switches, when configured to be “on,” act as diodes. Designing circuits for the MLA is significantly different than for a programmable logic array, which requires an OR and an AND plane. We have preliminary designs for a “standard cell” library using nanoBlocks (e.g., AND, OR, XOR, and ADDER).

Fig. 50 shows the implementation of an AND gate, while Fig. 51 shows the implementation for a half-adder. On the top part of Fig. 51 is a schematic of the portion of the circuit used to generate the sum output. This circuit, which is the XOR of A and B, is typical of diode-resistor logic. For example, if A is high and B is low, then their complements ($\overline{A}$ and $\overline{B}$) are low and high respectively. As a result, diodes 1, 2, 5, and 6 will be reverse-biased and not conducting. Diode 8 will be forward-biased and will pull the line labeled “red” in the figure down close to a logic low. This makes diode 4 forward-biased. By manufacturing the resistors appropriately (i.e., resistors attached to V_{dd} have smaller impedances than those attached to Gnd) most of the voltage drop occurs across R1, resulting in S being high. If A and B are both low, then diodes 2, 4, 5, and 7 are back-biased. This isolates S from V_{dd} and makes it low.

The MLA computes logic functions and routes signals using diode-resistor logic. The benefit of this scheme is that we can construct it by directed assembly, but the drawback is that the signal is degraded every time it goes through a configurable switch. In order to restore signals to proper logic values without using CMOS gates, we will use the molecular latch described in Section 8.3.

The organization of the MLA supports the implementation of logic functions using either two-level logic or arbitrary logic combinations.[31] If a two-level representation is used, for example, a product of sums representation, then the square shape of the MLA and the fact that all the power connections are on one side and all the connections to ground are on another will reduce the logic density that can be realized in a given area. This is offset by three factors. First, it will be easier to fabricate a symmetric MLA. Second, the extra freedom to place product terms or sum terms anywhere in the array increases defect tolerance. Third, as we will see below, reducing the number of different connections to the CMOS substrate increases the overall density of the nanoscale components.

Arbitrary logic representations use few of the cross-points on any wire. As the size of the MLA increases, the overhead of the micronscale connections

[31] Refer back to Section 5.2 for definitions of these terms.

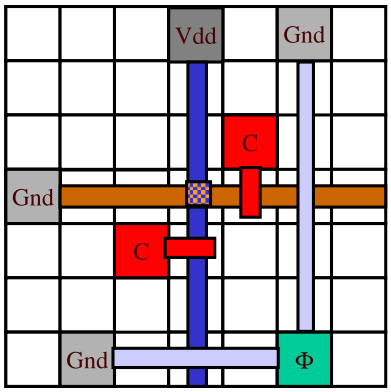


Figure 52: An example layout of a nanoBlock and its associated CMOS support. The long thin rectangles represent nanowires. The thicker squares represent CMOS metal.

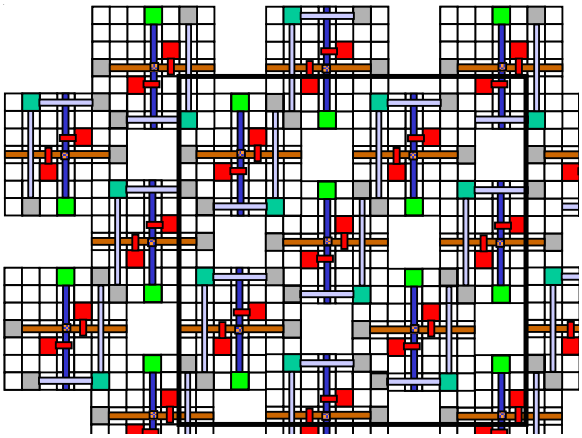


Figure 53: How nanoblocks tile together.

decreases, but the density of useful connections per wire decreases. The goal of the designer is to choose the size of the MLA so that the overall logic density-per-unit area is highest. In addition to optimizing the cross-points on each wire, the designer must keep in mind that a particular circuit cannot include an arbitrary number of diodes. Each time a signal passes through a diode the signal is degraded. Therefore, the total number of diodes that can be traversed is limited by the number of diodes that can be traversed before a logical value on the wire can no longer be distinguished from its opposite value.

The layout of the MLA and of the switch block makes rerouting easy in the presence of faults. By examining Fig. 51, one can see that a bad switch is easily avoided by swapping wires that only carry internal values. In fact, the rows can be moved anywhere within the block without affecting the circuit, which makes defect tolerance significantly easier than with CMOS.[32] The number of possible ways to arrange the columns/rows in the MLA combined with the configurable cross-bar implemented by the switch block makes the entire design robust to defects in either the switch block or the MLA.

Notice that all the connections between the CMOS layer and the nanoBlock occur either between groups of wires or with a wire that is seperated from all the other components. This improves the device density of the fabric. To achieve a specific functionality, the cross-points are configured to be either open connections or to be diodes.

Fig. 52 shows one possible way that the nanoscale components of the nanoBlock might be connected to a CMOS substrate. The filled-in squares represent connections to the underlying substrate. The rectangles represent nanoscale wires. The power (V_{dd}) and ground (Gnd) connections are all to sets of wires. The power connection is for pull-up resistors in the MLA, the ground connection on the left is for the pull-down resistors in the MLA, and the two other connections are for the connections to the molecular latch. The other end of the molecular latch wires connect the clock pad (Φ). Finally, the two CMOS connections labeled “C” are for the configuration pad that connects to the single-wire demux. Notice that only a tiny fraction of the entire area is taken up by the MLA itself (the checkerboard area in the middle of the nanoBlock.

The arrangement shown in Fig. 52 allows multiple nanoBlocks to tile the plane without violating any of the process rules used in constructing the CMOS pads. That is, no pads of different types share any sides or corners. Fig. 53 show how the nanoBlocks can be arranged into the diagonal patterns needed to form the switch blocks. The repeated pattern that can tessalate the plane is shown as the area surrounded by a thick black line covering eight nanoBlocks. The pattern covers a 20x20 region of CMOS line widths

[32] This is because nanowires have several orders of magnitude less resistance than switches. Therefore, the timing of a block is unaffected by small permutations in the layout.

and contains eight nanoblocks. Therefore, the area for one implemented nanoblock is 50 square minimum line widths. This is at least an order of magnitude smaller than a half-adder implemented solely in CMOS.

9.1.2 Configuration

The nanoFabric uses run-time reconfiguration for defect testing and to perform its intended function. Therefore, it is essential that the time to configure the fabric scale with the size of the device. There are two factors that contribute to the configuration time. The first factor is the time that it takes to download a configuration to the nanoFabric. The second factor is the time that it takes to distribute the configuration bits to the different regions of the nanoFabric. Configuration decoders are required to serialize the configuration process in each nanoBlock. (See Fig. 34 in Section 7.3.) To reduce the CMOS overhead, we intend to configure only one nanoBlock per cluster at a time. However, the fabric has been designed so that the clusters can be programmed in parallel. A very conservative estimate is that we can simultaneously configure one nanoBlock in each of 1000 clusters in parallel.

A molecular switch is configured when the voltage across the device is increased outside the normal operating range. Devices in the switch blocks can be configured directly by applying a voltage difference between the long intercluster lines. In order to achieve the densities presented above, it will also be necessary to develop a configuration approach for the switches in the MLA that is implemented with nanoscale components. In particular, a nanoscale decoder is required to address each intersection of the MLA independently. Instead of addressing each nanoscale wire separately in space, we address them separately in the time dimension. This slows down the configuration time but increases the device density.

Our preliminary calculations indicate that we can load the full nanoFabric, which is comprised of 10^9 configuration bits at a density of 10^{10} configuration bits/cm^2, in less than 1 second. This calculation is based on realistic assumptions that, on average, fewer than 10% of the bits are set ON and that the configurations are highly compressible [HLS99]. It also significant to note that it is not necessary to configure the full fabric for defect testing. Instead, we will configure only the portions under test.

As the configuration is loaded onto the fabric it will be used to configure the nanoBlocks. Using the configuration decoder this will require ≈ 300 cycles per nanoBlock, or less than 38K cycles per cluster. Therefore, the worst-case time to configure all the clusters at a very conservative 10 MHz requires 3 seconds.

9.1.3 Putting It All Together

The nanoFabric is a reconfigurable architecture built out of CMOS and CAEN technology. The support system, i.e., power, ground, clock, and configuration wires, I/O, and basic control, is implemented in CMOS. On top of the CMOS we construct the nanoBlocks and long-lines constructed out of chemically self-assembled nanoscale components.

Assuming a 100 nm CMOS process and 40 nm centers with 128 blocks to a cluster and 30 long-lines per channel, our design should yield 200M blocks/cm^2), requiring 2^{10} configuration bits. (If, instead of molecular latches, transistors were used for signal restoration then with 40 nm centers for the nanoscale wires the density is reduced by a factor of 10.)

SPICE simulations show that a nanoBlock configured to act as a half-adder can conservatively operate at 100MHz. Preliminary calculations show that the fabric as a whole will have a static power dissipation of ≈ 1.2watts and dynamic power consumption of $\approx .4$watts at 100Mhz.

The nanoFabric assumes a fairly simple fabrication process but still requires the parallel alignment of nanoscale wires and the ability to attach those wires to photolithographically-created pads on a silicon substrate. It overcomes the lack of arbitrary patterning in the fabrication methods with postfabrication computation, which will both aid in defect tolerance and allow the desired functionality to be configured into the device.

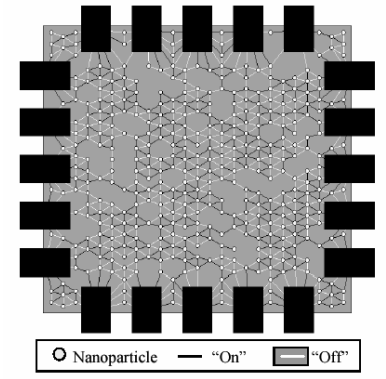


Figure 54: A representation of a Nanocell. The links between particles are implicitly created due to the proximity of the molecules. Reprinted with permission from [Hus02], S. M. Husband., Programming the Nanocell, a Random Array of Molecules, Ph.D. Thesis, Rice University, 2002.

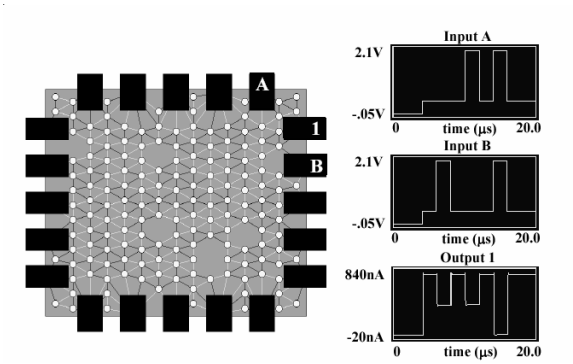


Figure 55: A Nanocell programmed to implement a NAND-gate in its upper-right corner. Reprinted with permission from [Hus02], S. M. Husband., Programming the Nanocell, a Random Array of Molecules, Ph.D. Thesis, Rice University, 2002.

9.2 Random Approach

The random class of architectures requires less precise fabrication methods than nanoFabric. This class of architectures essentially requires only fabrication methods to randomly deposit nanoscale particles on a silicon substrate. However, it requires even more postfabrication computation to construct useful computing devices.

Here we describe an example from this class of architecture proposed by a team at Rice and Yale. The basic component of this architecture is the Nanocell [Hus02, HHP+01], a small lithographically-defined area filled with nanoparticles and molecules. The nanoparticles are assembled in the interior using a self-assembled monolayer (see Fig. 54). Molecules that behave like molecular switches and also exhibit NDR behaviors are deposited on the nanoparticles. Along the perimeter of the nanocell are I/O pins which connect the nanoparticles in the interior of the cell with the rest of the chip, which is traditional CMOS created using lithographic techniques. The density of particles and deposited molecules ensures that there will be some connections between the particles themselves and the micronscale leads connected around the perimeter of the cell. The resulting ensemble forms a programmable electrical network.

Each nanocell will have a different arrangement of particles and, as a result, will possibly have different functionality. Before a nanocell can be used in a circuit, its possible set of functionalities will have to be determined and then it will have to be programmed to implement the desired functionality. Husband [Hus02] proposes a genetic algorithm which can be used to find the functionality of each cell and to train the cell to implement one of its possible functions. The algorithm depends on being able to set the state of each molecule individually. Fig. 55 shows an example of a nanocell that was trained to implement a NAND gate. Once the cells are programmed, they need to be connected. While some nanocells have been shown in simulation to implement restoring logic, they do not provide I/O isolation. Therefore, this architecture requires that the interconnect in a circuit be implemented at least partially in CMOS. Alternatively, molecular latches can be used to restore signals and provide I/O-isolation as discussed in Section 8.3.

The nanocell is defect tolerant but results in rather poor logic density. It is, by its nature, defect tolerant in that no predetermined functionality is assumed, rather it is discovered. However, whether the functionality needed to implement a useful circuit can be found in a reasonable amount of time is an open question. The density of a nanocell-based computer is determined by the size and functionality of the nanocell, the supporting interconnect, and

any logic needed for signal restoration or I/O isolation. Even if the supporting logic takes no area and the interconnect is minimal, the density is low because the I/O pins to the nanocell are fabricated using lithography. If the lithographic pitch is P_l, then a nanocell with 20 I/O pins requires $(12P_l)^2$ units of area. A NAND gate implemented in CMOS requires $\sim 35P_l$ units of area. In other words, a nanocell implementing four NAND gates will be only slightly denser than CMOS. CMOS has the advantage that no static power is consumed. If NMOS-style circuits are used, then the lithographic approach is significantly denser than the nanocell. If nanocells can implement more complicated functions (e.g., a half-adder), the nanocell may enjoy a slight density advantage.

The nanocell architecture is an example of a molecular electronic architecture that presupposes only the simplest assembly primitives. It is defect tolerant and reconfigurable. However, it requires significant programming effort to overcome the randomness inherent in its fabrication. It requires significant micronscale functionality during circuit operation, in particular, it requires micronscale wires and potentially micronscale devices, to pass signals from one nanocell to another.

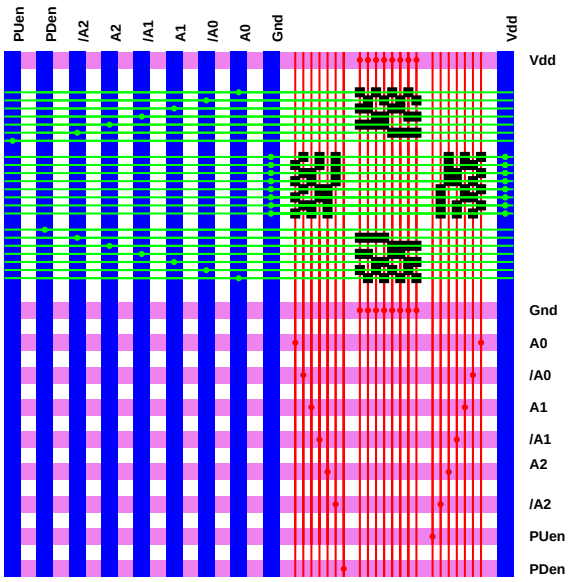


Figure 56: A nanoscale array surrounded by four decoders and the micronscale wiring necessary to program it. Reprinted with permission from [DeH02] A. DeHon, in "Proceedings of the First Workshop on Non-Silicon Computation," 2002.

9.3 Quasi-Regular Approaches

We have already seen an instance of a quasi-regular architecture, the nanoFabric. It assumes that the fabrication primitives can be used to create two-dimensional meshes of wires. It further assumes that some kind of active device can be made at the intersection of two wires and that the wires can be bonded to lithographically created contacts. It does not assume that the wires in a mesh can be deterministically ordered. In spite of these seemingly simple primitives, the nanoFabric and other architectures in this class are surprisingly powerful. Here we discuss briefly an additional example from this class.

DeHon [DeH02] proposes an architecture based around arrays of silicon nanowire FETs. The arrays are arranged so that all signals in a circuit can be transmitted over nanoscale wires. At least some of the active elements in the arrays are reconfigurable, which supports defect tolerance and the creation of circuits after the device is fabricated. The nanoscale components are surrounded by lithographically created wires and circuits that are used only to support reconfiguration and to supply global signals (e.g., clock, power, and ground).

Fig. 56 shows a possible fundamental building block of this architecture. Each nanoscale array is surrounded by a set of four decoders. The decoders connect the nanoscale wires to address lines that are micronscale. The address lines allow any individual nanowire to be selected. By selecting a particular row and column, the switch at the cross-point can be configured. The decoders can also be used to provide pull-up or pull-down resistors for either diode-resistor logic or NMOS-style transistor logic. The decoders allow $O(\log_2 n)$ micronscale wires to address n nanoscale wires. The total area for an array is therefore $((A+4)P_l + (2A+n)P_n)^2$, where n is the number of wires in each dimension of the nanoscale array, A is the number of address lines used to address the wires in the nanoscale array $A \geq \log_2 n$), and P_n is the pitch of the nanoscale wires. In this arrangement with perfect decoders (i.e., $A = \log_2 n$) and a 10:1 ratio between the

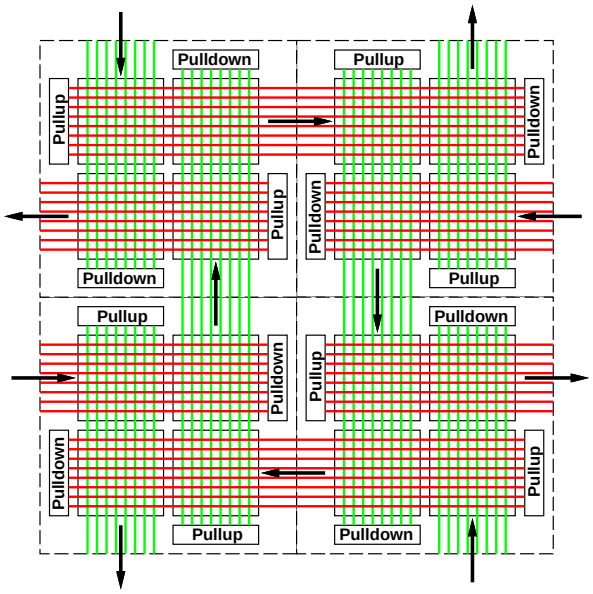


Figure 57: Arrangement of NOR only planes. Reprinted with permission from [DeH02] A. DeHon, in "Proceedings of the First Workshop on Non-Silicon Computation," 2002.

lithographic pitch and the nanowire pitch, then the support logic takes less than half the area when n reaches several hundred. Either nanoimprinting or self-assembly (see Section 7.3) can be used to construct the decoders. If fault tolerance is needed, or if the self-assembly method described in [WK01] is used, then A will be several times $\log_2 n$ requiring arrays of several thousand nanoscale wires before the overhead is less than half.

When we compare the density of this approach to the nanoFabric, it is important to look beyond the raw density of nanoscale cross-points. The nanoFabric also devotes approximately 50% of its area to support logic when n is several hundred. However, the single-wire demux would probably not scale to that large of an array. The more important issue is whether the large arrays can be effectively utilized when logic is mapped to them. If a nonrestoring logic is used, then very few of the cross-points in either array would be used, effectively reducing the density of the nanoscale portion of the architecture significantly. Therefore, several factors need to be considered. First, does the micronscale-to-nanoscale addressing mechanism scale to the desired array size? Second, can the nanoscale array be effectively used to implement logical functions? Finally, will the array be sufficiently fault tolerant? We discuss this final issue in Section 10.

In order to implement logic in DeHon's architecture, the cross-points must be configurable. DeHon suggests two possible approaches. If the nanowire FETs are programmable, then the array shown in Fig. 56 can be replicated in groups of 16.

Fig. 57 shows the nanoscale portions of the arrays. The decoders used to program the configuration bits serve double duty, acting as either pull-up or pull-down resistors when the circuit is running. The black arrows show how the outputs from one set of NORs can be fed in as inputs to the orthogonal set of NOR wires. Take, for example, the second column from the right. It has pull-up resistors at the top, then there are four arrays running vertically, and finally a group of pull-down resistors is at the bottom. A nanowire FET running from the pull-up to the pull-down implements a NOR gate. The inputs to the gate are from the top left or the top right. The outputs go either to the bottom right or the bottom left. Therefore, there is substantial flexibility in routing to and from each gate. Furthermore, all the connections between gates use nanoscale wires.

If the nanowire FETs are not configurable, then half the arrays can be replaced with configurable diodes that can implement an OR plane. This can be combined with the nonconfigurable NOR gates to enable a PAL-like approach to creating logic. The OR gates will be programmable, while the NOR gates can provide I/O isolation, inversion, and signal restoration.

This architecture, like the nanoFabric, assumes a moderate amount of precision and requires a moderate amount of postfabrication computation in order to construct useful computing devices. There are several other architectures in this class. In 1998, a team from HP and UCLA proposed a defect-tolerant architecture roughly based on the Teramac [HKSW98]. Their overall approach was the inspiration behind the nanoFabric. A team at AFRL proposed an architecture that is a cross between the quasi-regular and the deterministic approaches. At the micronscale it is regular, but at the nanoscale it requires some arbitrary patterning [LDK01, LDK00].

9.4 Deterministic Approach

As one would expect, there are few proposed architectures that require complete deterministic control at the nanoscale. A team at Mitre made early proposals for multifunctional molecules. Their approach is to exploit the huge number of molecules possible and to design molecules that can act as AND gates or even half-adders [EL00].

A completely different approach is found in the literature on quantum cellular automata (QCA). QCAs move information around using charge instead of current [Por98, NK01, FRJ+02]. There are many advantages to using QCAs, particularly with respect to power. However, the individual cell in a QCA provides no isolation or gain. The designers of QCA systems address this problem by adding a system clock. The clock provides isolation and signal restoration. Finally, the arrangement of the cells in a QCA, with respect to each other and the clock, will require great precision. It does not appear that QCA circuits offer any advantage in terms of manufacturability over CMOS. Recently, however, there have been proposals for reconfigurable QCAs [NRK02] that could possibly reduce the amount of manufacturing precision required.

9.5 Architectural Constraints

In summary, all the successful architectural approaches for molecular electronics must contend with atomic scale effects. The small size of the individual components guarantees that there will be some variability, some defects, and some constraint on the kinds of patterns that can be economically created. In addition, to interface the atomic scale electronics to the outside world implies some kind of size matching problem. Finally, the small size also entails that there will be huge numbers of devices. This results in issues relating to scalability.

10 Defect Tolerance

The nanoFabric is defect tolerant because it is regular, highly configurable, fine-grained, and, has a rich interconnect.[33] The regularity allows us to choose where a particular function is implemented. The configurability allows us to pick which nanowires, nanoBlocks, or parts of a nanoBlock will implement a particular circuit. The fine-grained nature of the device combined with the local nature of the interconnect reduces the impact of a defect to only a small portion of the fabric (or even a small portion of a single nanoBlock). Finally, the rich interconnect allows us to choose among many paths in implementing a circuit. Therefore, with a defect map we can create working circuits on a defective fabric. Defect discovery relies on the fact that we can configure the nanoFabric to implement any circuit, which implies that we can configure the nanoFabric to test its own resources.

The key difficulty in testing the nanoFabric (or any FPGA) is that it is not possible to test the individual components in isolation. Researchers on the Teramac project [ACC+95] faced similar issues. They devised an algorithm that allowed the Teramac, in conjunction with an outside host, to test itself [CAC+97, CAC+96]. Despite the fact that over 75% of the chips in the Teramac have defects, the Teramac is still used today. The basis of the defect mapping algorithm is to configure a set of devices to act as tester circuits. These circuits (e.g., linear-feedback shift-registers) will report a result which, if correct, indicates with high probability that the devices they are made from are fault-free.

In some sense, Teramac introduces a new manufacturing paradigm, one which trades off complexity at manufacturing time with postfabrication programming. The reduction in manufacturing time complexity makes reconfigurable fabrics a particularly attractive architecture for CAEN-based circuits, since directed self-assembly will most easily result in highly regular, homogeneous structures. We expect that the fabrication process for these fabrics will be followed by a testing phase, where a defect map will be created and shipped with the fabric. The defect

[33] Defect tolerance through configuration also depends on short circuits being significantly less likely to occur than stuck-open faults. The synthesis techniques should be biased to increase the likelihood of a stuck-open fault at the expense of potentially introducing more total faults.

map will then be used by compilers to route around the defects.

The key problem is then locating the defects: once these locations are known, routing around them should be straightforward enough. In general, any methodology for locating these defects should not require direct access to individual components and should scale slowly with both defect density and fabric size.

We outline here an approach to defect mapping that extends the techniques used for the Teramac. While conceptually similar to these earlier techniques, the approach described here makes some key contributions that enable mapping of the much higher defect rates expected in CAEN-based fabrics.

Modern DRAM and SRAM chips and FPGAs are able to tolerate some defects by having redundancy built into them: for instance, a row containing a defect might be replaced with a spare row after fabrication. With CAEN fabrics, this will not be possible: it is unlikely that a row or column of any appreciable size will be defect free. Moreover, CAEN-based devices are being projected as a replacement not just for memories but for logic as well, where simple row replacement will not work since logic is less regular.

There is a large body of work in statistics and information theory on techniques for finding subsets of a population all members of which satisfy a given property (in our case, this translates to finding defect-free devices from among a large number, some of which may be defective). Various flavors of this technique, called *group testing*, have been applied to a variety of problems [Dor43, SG59, Wol85, KBT96]. However, none have constraints as demanding as those of CAEN-based assembly.

Problems similar to this have been addressed in the domain of custom computing systems. For example, the Piperench reconfigurable processor [SKG00], and more notably the Teramac custom computer [CAC+97, HKSW98], had a notion of testing, defect-mapping, and defect-avoidance built into them. Assembly was followed by a testing phase where the defects in the FPGAs were identified and mapped. Compilers for generating FPGA configurations then use this defect map to avoid these defects. The testing strategy we outline is similar to the one used for the Teramac. However, the problem we address is significantly harder because the Teramac used CMOS devices whose defect rates are much lower than those predicted for nanoFabrics.

An alternative approach to achieve defect tolerance would be to use techniques developed for fault-tolerant circuit design (e.g., [Pip90, Spi96]). Such circuit designs range from simple ones involving *triple-mode redundancy* to more complex circuits that perform computation in an alternative, sparse code space, so that a certain number of errors in the output (up to half the minimum distance between any two code words) can be corrected. Such techniques work reliably only if the number of defects is below a certain hard threshold, which CAEN is likely to exceed. Furthermore, even the best techniques for fault tolerance available today require a significant amount of extra physical resources, and also result in a (non-negligible) slow-down of the computation.

10.1 Methodology

One approach to defect detection in molecular-scale reconfigurable circuits consists of configuring the components on the fabric[34] into test circuits, which can report on the presence or absence of defects in their constituent components. Each component is made a part of many different test circuits and information about the error status of each of those circuits is collected. This information is used to deduce and confirm the exact location of the defects.

As an example, consider the situation in Fig. 58. Five components are configured into one test circuit that computes a simple mathematical function. This function is such that defects in one or more circuit components would cause the answer to diverge from the correct value. Therefore, by comparing the circuit's output with the correct answer, the presence or absence of any defects in the circuit components can

[34] We are deliberately leaving the meaning of "component" unspecified. It will depend on the final design of the fabric: a component may be one or more simple logic gates, or a look-up table implementing an arbitrary logic function. Also, the on-fabric interconnects will also be "components" in the sense that they may also be defective.

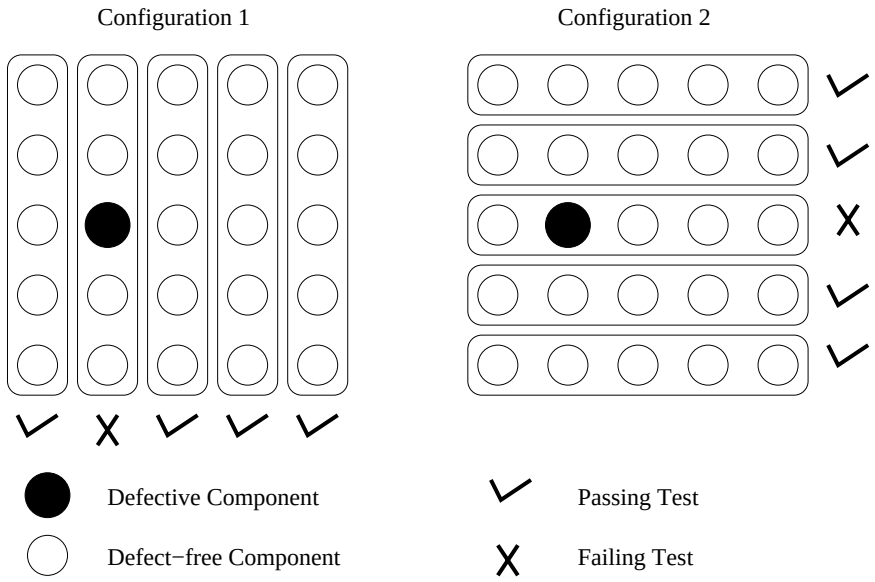


Figure 58: An example showing how a defective component is located using two different test circuit configurations. The components within one rectangular block are part of one test circuit.

be detected. In the first run, the components are configured vertically and test circuit 2 detects a defect. In the next run, the components are configured horizontally and test circuit 3 fails. Since no other errors are detected, we can conclude that the component at the intersection of these two circuits is defective, and all others are good.

Since the tester will not have access to individual fabric components, the test circuits will be large, consisting of tens and perhaps even hundreds of components. With high defect rates, each circuit will on average have multiple defective components. This will considerably complicate the simple picture presented in the example above. In particular, test circuits which give information only about the presence or absence of defects (such as the ones used above) will be useless: almost each and every test circuit will report the presence of defects. The key idea we use here is more powerful test circuits: circuits that return more information about the defects in their components, such as a count of the defects. An example would be a circuit that computes a mathematical function whose output will deviate from the correct value if any of the circuit's components are defective. If the amount of this deviation deterministically depends on the number of defective components, then a comparison of the circuit's output with the correct result can tell us the number of defects present in the circuit.

Using such defect-counting circuits, Mishra and Goldstein [MG02] propose splitting the process of defect mapping into two phases: a *probability-assignment phase* and a *defect-location phase*. The probability-assignment phase attempts to separate the components in the fabric into two groups: those that are probably good and those that are probably bad. The former will have an expected defect density that is low enough so that in the defect-location phase, we can use circuits that return 0-1 information about the presence of defects to pinpoint them.

The first phase, that of probability assignment, works as follows:

1. The components are configured into test circuits in many different orientations (for example, vertically, horizontally, and diagonally), and defect counts for all the test circuits are noted.

2. Given these counts for all the circuits, simple Bayesian analysis is used to find the probability that any particular component is good or bad.

3. The components with a low probability of being good are discarded and this whole process is repeated. This is done a small number of times, so that as many of the defective components as possible are eliminated.

At the end of this process, two groups of components are obtained: those with a high probability of being good and those with a high probability of being bad. The latter are discarded. In the former, the fraction of faulty components is expected to be at such a level that a significant number of defect-free circuits can be found. Therefore, the second phase, that of defect location, proceeds as follows:

1. Test circuits providing 0-1 defect information are run on the reduced set of components. If a circuit is found to be defect-free, all its components are marked as good. This is done again for a number of test circuit orientations.

2. For a high yield, some of the components discarded in the probability assignment phase can be added back and step 1 repeated.

3. Finally, all the components marked as good are declared good, and the others are declared bad.

There are a number of points to note here:

1. The quality of the results will depend on the quality of the test circuit. In particular, we assume that our counter circuits can count defects only up to a certain threshold (this will likely be the case, for example, if *error-correction* techniques such as Reed-Solomon codes [RS60] are used for test circuit design). Therefore, a higher threshold will correspond to better quality results.

2. A number of good components may be marked bad, particularly if a low-threshold test circuit is used. This is the *waste* of the procedure. It should be noted that it is never the case that a bad component is marked good.

3. With the testing strategy as described above, it will not be possible to completely determine the test circuits *a priori*, and some feedback dependent configuration on the part of the tester will be required. The feedback will be limited to routing the predetermined test circuit configurations around the discarded components.

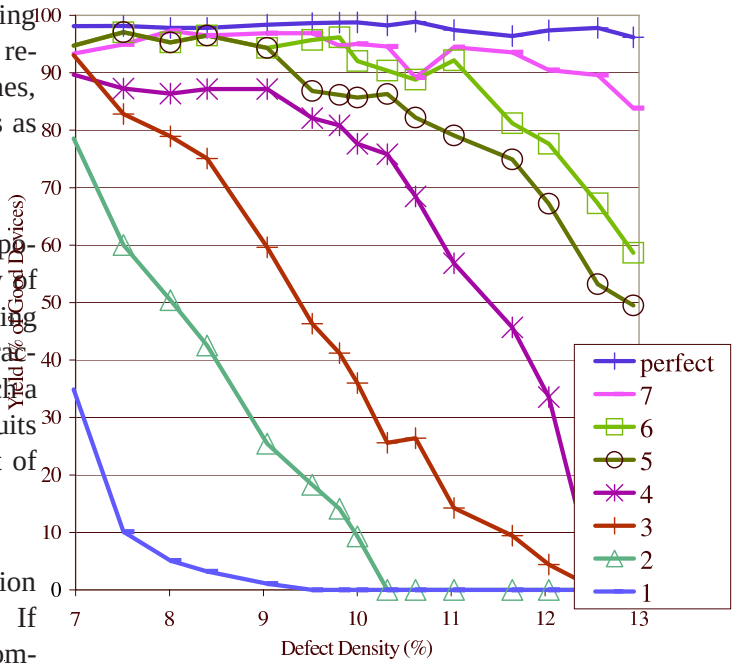


Figure 59: Yields achieved by varying the defect densities and the number of defects our test circuits could count. The *X* axis represents the defect density of the fabric, the *Y* axis shows the yield achieved (or, in other words, the fraction of the fabric's defect-free components that are identified as such), and each line represents a counter that can count defects up to a different threshold.

To test the effectiveness of this procedure and to measure the impact of the defect-counting threshold on the results, we ran a number of simulations; the results are presented in Fig. 59. From our results, it is apparent that it is possible to achieve high yields even with test circuits that can count a small number of defects. For example, for densities less than 10%, a test circuit that could count up to four errors achieved yields of over 80%. With more powerful test circuits, yields of over 95% are achievable.

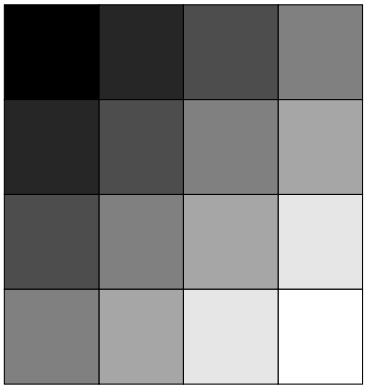


Figure 60: A schematic representation of how testing will proceed in a wavelike manner through the fabric. The black area is tested and configured as a tester by the external tester. Each darker shaded area then tests and configures a lighter-shaded neighbor. For large fabrics, multiple such waves may grow out from different externally tested areas.

10.2 Scaling with Fabric Size

A short testing time is crucial to maintain the low cost of CAEN fabrics, and therefore it needs to be ensured that testing proceeds quickly and needs minimal support from an external tester. Our testing strategy currently requires time proportional to n for testing a fabric of $n \times n$ components, if a test circuit of size n is being used. To speed testing up even further, reconfigurability of the fabric can be leveraged in the following ways:

1. Once a part of the fabric is tested and defect-mapped, it can be configured to act as a tester for the other parts, therefore reducing the time required on the external tester drastically.

2. Once the tester is configured onto the fabric, there is nothing to prevent us from having multiple testers active simultaneously. In such a scenario, each tested area tests its adjacent ones and the testing can proceed in a wave through the fabric (see Fig. 60).

Therefore, the reconfigurability of the fabric helps reduce the time on the external tester and also the total testing time by a significant amount.

Once a defect map has been generated, the fabric can be used to implement arbitrary circuits. The architecture of the nanoBlock supports full utilization of the device even in the presence of a significant number of defects. Due to the way we map logic to wires and switches, only about 20% of the switches will be in use at any one time. Since the internal lines in a nanoBlock are completely interchangeable, we generally should be able to arrange the switches that need to be configured in the ON state to be on wires which avoid the defects.

While the molecules are expected to be robust over time, inevitably new defects will occur over time. Finding these defects, however, will be significantly easier than doing the original defect mapping because the unknown defect density will be very low.

11 Using Molecular Architectures

There are two scenarios in which nanoFabrics can be used: as factory-programmable devices configured by the manufacturer to emulate a processor or other computing device, and as reconfigurable computing devices.

In a manufacturer-configured device, user applications treat the device as a fixed processor (or potentially as a small number of different processors). Processor designers will use traditional CAD tools to create designs using standard cell libraries. These designs will then be mapped to a particular chip, taking into account the chip's defects. A finished product is therefore a nanoFabric chip and a ROM containing the configuration for that chip. In this mode, the configurability of the nanoFabric is used only to accommodate a defect-prone manufacturing process. While this provides the significant benefits of reduced cost and increased densities, it ignores much of the potential in a nanoFabric. Since defect tolerance requires that a nanoFabric be reconfigurable, why not exploit the reconfigurability to build application-specific processors?

Reconfigurable fabrics offer high performance and efficiency because they can implement hardware matched to each application. Further, the configurations are created at compile-time, eliminating the

need for complex control circuitry. Research has already shown that the ability of the compiler to examine the entire application gives a reconfigurable device efficiency advantages because it can:

- exploit all of an application's parallelism: task-based, data, instruction-level, pipeline, and bit-level,
- create customized function units,
- eliminate a significant amount of control circuitry,
- reduce memory bandwidth requirements,
- size function units to the application's natural word size,
- use partial evaluation and constant propagation to reduce the complexity of operations.

However, this extra performance comes at the cost of significant work by the compiler. A conservative estimate for the number of configurable switches in a 1cm^2 nanoFabric, including all the overhead for buffers, clock, power, etc., is on the order of 10^{11}. Even assuming that a compiler manipulates only standard cells, the complexity of mapping a circuit design to a nanoFabric will be huge, creating a compilation scalability problem. Traditional approaches to place-and-route in particular will not scale to devices with billions of wires and devices.

In order to exploit the advantages listed above, we propose a hierarchy of abstract machines that will hide complexity and provide an intellectual lever for compiler designers while preserving the advantages of reconfigurable fabrics. At the highest level is a split-phase abstract machine (SAM), which allows a program to be broken up into autonomous units. Each unit can be individually placed and routed and then the resulting netlist of preplaced and routed units can be placed. This hierarchical approach will allow the CAD tools to scale.

11.1 Split-Phase Abstract Machine

The compilation process starts by partitioning the application into a collection of threads. Each thread is a sequence of instructions ending in a split-phase operation. An operation is deemed to be a split-phase operation if it has an unpredictable latency. For example, memory references and procedure calls are all split-phase operations. Therefore, each thread, similar in spirit to a threaded abstract machine (TAM) thread [CGSvE93], communicates with other threads asynchronously using split-phase operations. This partitioning allows us to virtualize the hardware (by swapping threads in and out as necessary), allows the CAD tools to concentrate on mapping small isolated netlists, and has all the mechanisms required to support thread-based parallelism.

Unlike a traditional thread model, where a thread is associated with a processor when executing, each SAM thread will be a custom "processor." While it is possible for a thread to be complex and load "instructions" from its local store, the intention is that it remains fairly simple, implementing only a small piece of a procedure. This allows the threads to act either in parallel or as a series of sequential processes. It also reduces the number of timing constraints on the system, which is vital for increasing defect tolerance and decreasing compiler complexity.

The SAM model is a simplification of TAM. A SAM thread/processor is similar to a single-threaded codeblock in TAM, and memory operations in SAM are similar to memory operations in Split-C [CDG+93]. In a sense, SAM will implement (in reconfigurable hardware) active messages [vECGS92] for all interprocessor communications. This model is powerful enough to support multithreading. As the compiler technology becomes more mature, the inherently parallel nature of the model can be exploited.

While SAM can support parallel computation, a parallelizing compiler is not necessary. The performance of this model rests on the ability to create custom processors. A compiler could (and in its first incarnations will) construct machines in which only one processor is active at a time.

This also creates the opportunity to virtualize the entire system. Imagine each configuration of a processor as an ultrawide instruction that can be loaded as needed to virtualize the hardware. The bandwidth to load these instructions is available since CAEN

devices can implement very dense customized memories. Later, as the compiler technology becomes more mature, the inherently parallel nature of the model can be exploited.

The SAM model explicitly hides many important details. For example, it addresses neither dynamic routing of messages nor allocation of stacks to the threads. Once an application has been turned into a set of cooperating SAM threads, it is mapped to a more concrete architectural model that takes these issues into account. The mapping process will, when required, assign local stacks to threads, insert circuits to handle stack overflow, and create a network for routing messages with run-time computed addresses. For messages with addresses known at compile-time, it will route signals directly.

12 Conclusions

Recent advances in molecular electronics, combined with increased challenges in semiconductor manufacturing, create a new opportunity for computer architects—the opportunity to recreate computer architecture from the ground up. New device characteristics require us to rethink the basic abstraction of the transistor. New fabrication methods require us to rethink the basic circuit abstraction. The scale of the devices and wires allows us to rethink our basic approach to designing computing systems. On the one hand, the scale enables huge computing systems with billions of components. On the other hand, the scale forces us to rethink the meaning of a working system; it must be a reliable system made from unreliable components.

The main challenge facing designers today is in reexamining and redefining the core abstractions that have enabled us to successfully design and implement computers. One approach to computer architecture is to view it as a hierarchy of abstractions. The bottom layer of one possible hierarchy is the device—currently the transistor. Built on the device layer is the abstraction of the logic gate (e.g., a NAND gate). On top of this, we have circuits and then function units (e.g., adders and state machines). Above function units, we have the instruction set architecture that defines the interface between the hardware and the software in a computing system. Above these layers there are computing models, algorithms, and programming languages. A hierarchy of abstractions implies a contract between layers in the hierarchy. The contract allows one to change the internals of any particular layer without adversely affecting the entire system. It allows many people to simultaneously work on advancing the state of the art. The advances in computing power over the last few decades are an excellent example of the effectiveness of this hierarchy of abstractions—almost without exception, advances have been made without having to break the abstractions.

On the other hand, sometimes one must intentionally break the contracts, either by piercing the abstractions or by creating new ones. When done carefully the results can be dramatic. Molecular computing may be a case where a reexamination of the layers of abstraction is required. For example, as we have shown earlier, the fundamental device abstraction may be too rigid for nanometer-scale devices. Decomposing the device abstraction into four components—a switch, an isolator, a restorer, and a memory (SirM)—provides device designers and circuit designers alike with a new degree of freedom.

SirM is an example of the easiest kind of change to the abstraction hierarchy. It can be thought of as a new layer between the new physical devices and today's abstraction of a transistor. In other words, a "transistor"-like contract can be established by combining the proper switch, isolator, and restorer devices all together. Therefore, we are able to maintain the current abstraction, providing backward compatibility for tools and ideas while providing the opportunity to pierce the transistor abstraction when necessary. This provides the design community with an incremental path toward harnessing the underlying devices directly. Furthermore, it oes not change any of the contracts with other layers in the hierarchy.

A more difficult change to work into the abstraction hierarchy is wrought by the level of manufacturing defects and transient faults expected to arise from using nanometer scale devices. It is probably not reasonable to maintain the defect- and fault-free clause in the contract expected from today's devices

and circuits. We have argued in this chapter that exposing the defects in the manufacturing process to the higher levels in the hierarchy may reduce the total cost of the system. It may also be significantly easier to mask defects and faults from the end-user if each layer is expected to do some of the work. With respect to defect tolerance, a combined attack at the levels of circuits, architecture, and tools may be the only scalable approach. Reconfigurable architectures allow one to configure a hardware system to implement a particular circuit. By avoiding the faulty components, one may create reliable systems using unreliable components. This requires changes in architecture, circuit design, testing, and the tools used to create the circuits from an engineer's description.

One way to reduce the impact of a change in the abstraction hierarchy is to encapsulate the change in a tool. For example, one can present the abstraction of a fault-free manufacturing process if the tools that convert circuit descriptions to circuit layout implement all the changes necessary to perform defect tolerance. By encapsulating the details in a tool, we obtain the intellectual leverage of hiding the implementation details while at the same obtaining the advantage of piercing the abstraction layer.

We believe that successfully harnessing the power of molecular computing requires us to rethink several key abstractions: the transistor, the circuit, and the ISA. As we noted earlier, we feel that the abstraction of the transistor should be replaced by SirM. The abstraction that a user's desired circuit is created at manufacturing time needs to be replaced by the ability to configure circuits at run-time. Finally, the abstraction of an ISA needs to be replaced with a more flexible hardware/software interface implemented in the compiler.

In just the last couple of years, there has been a convergence in molecular-scale architectures towards reconfigurable platforms. In the nanoFabric, the main computing element is a molecular-based reconfigurable switch. We exploit the reconfigurable nature of the nanoFabric to provide defect tolerance and to support reconfigurable computing. Reconfigurable computing not only offers the promise of increased performance but it also amortizes the cost of chip manufacturing across many users by allowing circuits to be configured postfabrication. The molecular-based switch eliminates much of the overhead needed to support reconfiguration—the switch holds its own state and can be programmed without extra wires, making nanoFabrics ideal for reconfigurable computing.

The notion of an ISA is very successful in hiding the details of a processor implementation from the user of a processor. However, it also limits the ability of a compiler to take full advantage of the processor. ISAs were created when computers were programmed by hand. They were therefore designed to be easily understandable by humans. Today, computers are programmed using high-level languages that are then translated by compilers into the machine level instructions that control the computer. Therefore, there is less of a need to make an ISA concise and easy to understand. We argue that harnessing the plentiful resources that will become available through molecular computing will require exposing more of the underlying computational structure to the compiler. In addition, by exposing more of the underlying architecture, it should be easier for the compiler to support increased levels of defect and fault tolerance.

In conclusion, we believe that computer architecture is presently faced with an incredible opportunity, an opportunity to harness the power of molecular computing to create computing systems of unprecedented performance per dollar. To realize this promise, we need to rethink the basic abstractions that comprise a computer system. Just as current systems benefited greatly from the underlying technology—the silicon-based transistor—new systems will benefit from the molecular reconfigurable switch. This can serve as the basis for a reconfigurable computing system that can construct customized circuits tailored on the fly for every application.

References

[ACA01] P. Avouris, P. Collins, and M. Arnold. Engineering carbon nanotubes and nanotube circuits using electrical breakdown. *Science*, 292(5517):706–709, 2001.

[ACC+95] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, and G. Snider. Teramac–configurable custom computing. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 32–38, Napa, CA, April 1995.

[AHKB00] V. Agarwal, H. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 344–347, June 2000.

[Alt02] Altera Corporation. Apex device family. `http://www.altera.com/html/products/apex.html`, 2002.

[AN97] e. a. A. Nakajima. Room temperature operation of Si single-electron memory with self-aligned floating dot gate. *Appl. Phys. Lett*, 70(13):1742, 1997.

[AR74] A. Aviram and M. Ratner. Molecular rectifiers. *Chemical Physics Letters*, 29(2):277–283, Nov. 1974.

[AS62] I. Aleksander and R. Scarr. Tunnel devices as switching elements. *J British IRE*, 23(43):177–192, 1962.

[ASH+99] M. Abramovici, C. Strond, C. Hamilton, S. Wijesuriya, and V. Verma. Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications. In *International Test Conference*, pages 973–982, 1999.

[Aza00] K. Azar. The history of power dissipation. *Electronics Cooling Magazine*, 6(1):42–50, 2000.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BCKS94] R. Bissell, E. Cordova, A. Kaifer, and J. Sotddart. A chemically and electrochemically switchable molecular device. *Nature*, 369:133–7, May 12, 1994.

[Ben02] C. Bennett. Lecture at Carnegie Mellon, 2002.

[BHND01] A. Bachtold, P. Hadley, T. Nakanishi, and C. Dekker. Logic circuits with carbon nanotube transistors. *Science*, 294:1317–1320, November 2001.

[BMBG02] M. Budiu, M. Mishra, A. Bharambe, and S. C. Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *Proceedings of 2002 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2002.

[BR97] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pages 213–222, August 1997.

[BSWG00] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Euro-Par 2000 parallel processing: 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[CAC+96] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. The Teramac custom computer: Extending the limits with defect tolerance. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 2–10, 1996.

[CAC+97] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the Teramac custom computer. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 116–123, April 1997.

[Car63] J. Carroll. *Tunnel-Diode and Semiconductor Circuits*. McGraw-Hill, 1963.

[CDG+93] D. Culler *et al.* Parallel programming in split-c. In *Proceedings of the Supercomputing '93 Conference*, pages 262–273, Nov. 1993.

[CDHL00] Y. Cui, X. Duan, J. Hu, and C. Lieber. Doping and electrical transport in silicon nanowires. *Journal of Physical Chemistry B*, 104(22):5213–5216, 2000.

[CG01] R. Canaal and A. González. Reducing the complexity of the issue logic. In *Proceedings of the International Conference on Supercomputing (ICS-01)*, pages 312–320, June 2001.

[CGSvE93] D. Culler, S. Goldstein, K. Schauser, and T. v. Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[CH99] K. Compton and S. Hauck. Configurable computing: A survey of systems and software. Technical report, Northwestern University, Dept. of ECE, 1999.

[Col] R. Collins. Dr. Dobb's microprocessors resources: Intel secrets and bugs. http://www.x86.org/secrets/intelsecrets.htm.

[CR00] J. Chen and M. Reed. Molecular wires, switches and memories. In *Proceedings of International Conference on Molecular Electronics*, Kailua-Kona, HI, Dec. 2000.

[CRRT99] J. Chen, M. A. Reed, A. M. Rawlett, and J. M. Tour. Observation of a large on-off ratio and negative differential resistance in an electronic molecular switch. *Science*, 286(5444):1550–2, 1999.

[CWB+99] C. P. Collier *et al.* Electronically configurable molecular-based logic gates. *Science*, 285(5426):391–394, July 16 1999.

[CWR+00] J. Chen *et al.* Room-temperature negative differential resistance in nanoscale molecular junctions. *Applied Physics Letters*, 77(8):1224–1226, 2000.

[DeH96] A. DeHon. Dynamically programmable arrays: A step toward increased computational density. In *Proceedings*

of the 4th Canadian Workshop of Field-Programmable Devices, pages 47–54, May 1996.

[DeH99] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125–134, February 1999.

[DeH02] A. DeHon. Array-based architecture for molecular electronics. In *Proceedings of the First Workshop on Non-Silicon Computation (NSC-1)*, February 3, 2002.

[DMAA01] V. Derycke, R. Martel, J. Appenzeller, and P. Avouris. Carbon nanotube inter- and intramolecular logic gates. *Nano Letters*, 9(1):453–6, August 2001.

[Dor43] R. Dorfman. The detection of defective members of large populations. *Annals of Mathematical Statistics*, 14:436–440, Dec. 1943.

[Dre86] K. E. Drexler. *Engines of Creation*. Doubleday, New York, 1986.

[EL00] J. Ellenbogen and J. Love. Architectures for molecular electronic computers: 1. logic structures and an adder designed from molecular electronic diodes. *Proc. IEEE*, 88(3):386–426, 2000.

[Enc02] *New Encyclopaedia Britannica*. Encyclopaedia Britannica, 2002.

[ES82] L. Esaki and E. Soncini, editors. *Large Scale Integrated Circuits Technology*, pages 728–46. Nijhoff, 1982.

[FRJ+02] 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 *Proceedings of the First Workshop on Non-Silicon Computation*, February 2002.

[GB01] S. C. Goldstein and M. Budiu. NanoFabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 178–189, 2001.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GPKR97] K. F. Goser, C. Pacha, A. Kanstein, and M. L. Rossmann. Aspects of systems and circuits for nanoelectronics. *Proceedings of the IEEE*, 85(4):558–573, April 1997.

[GSB+00] S. C. Goldstein *et al.* PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[GSM+99] S. C. Goldstein *et al.* PipeRench: a coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 28–39, 1999.

[Ham50] R. W. Hamming. Error-detecting and error-correcting codes. *Bell Systems Technical Journal*, 29(2):147–160, 1950.

[Har01] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 2001)*, pages 642–649, Exhibit and Congress Center, Munich, Germany, March 2001.

[Hau98] S. Hauck. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[HB67] D. Holmes and P. Baynton. A monolithic gallium arsenide tunnel diode construction. In A. Strickland, editor, *Gallium Arsenide*, number 3 in Conference series, pages 236–240, London, 1967. Institute of Physics and Physical Society.

[HDC+01a] Y. Huang *et al.* Logic gates and computation from assembled nanowire building blocks. *Science*, 294(5545):1313–1317, November 9, 2001.

[HDC+01b] Y. Huang *et al.* Logic Gates and Computation from Assembled Nanowire Building Blocks. *Science*, 294(5545):1313–1317, 2001.

[HDW+01] Y. Huang, X. Duan, Q. Wei, , and C. Lieber. Directed assembly of one-dimensional nanostructures into functional networks. *Science*, 291(5504):630–633, January 2001.

[HHP+01] S. Husband *et al.* The Nanocell Approach to a Molecular Computer. Moletronics PI Meeting, Sante Fe, NM, March 2001.

[Hip01] K. Hipps. It's all about contacts. *Science*, 294(5542):536–537, October 19, 2001.

[HKSW98] J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280(5370), June 12 1998.

[HLS99] S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(8):1107–1113, August 1999.

[HMH01] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann, 1996.

[Hus02] S. M. Husband. *Programming the Nanocell, a Random Array of Molecules*. PhD thesis, Rice University, April 2002.

[KBT96] E. Knill, W. J. Bruno, and D. C. Torney. Nonadaptive group testing in the presence of errors. Technical

Report LAUR-95-2040, Los Alamos National Laboratory, Los Alamos, NM, September 1996.

[Kes97] S. Keshav. *An Engineering Approach to Computer Networks*. Addison-Wesley, 1997.

[Key85] R. W. Keyes. What makes a good computer device? *Science*, 230(4722):138–144, October 1985.

[KMM+01] N. I. Kovtyukhova *et al.* Layer-by-layer assembly of rectifying junctions in and on metal nanowires. *J. Phys. Chem. B*, 105:8762–8769, 2001.

[KMM+02] N. I. Kovtyukhova *et al.* Layer-by-layer self-assembly strategy for template synthesis of nanoscale devices. *Mater. Sci Eng. C*, 19:255–262, 2002.

[KS86] S. R. Kunkel and J. E. Smith. Optimal pipelining in supercomputers. In *Proceeding of the 13th International Symposium on Computer Architecture*, pages 404–411, 1986.

[KWC+00] T. Kamins *et al.* Chemical vapor deposition of Si nanowires nucleated by TiSi2 islands on Si. *Applied Physics Letters*, 76(5):562–4, 2000.

[Lan57] R. Landauer. Spatial variation of current and fields due to localized scatterers in metallic conduction. *IBM Journal of Research and Development*, 1(3), 1957.

[LC83] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[LCH00] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 22–36, 2000.

[LDK00] J. Lyke, G. Donohoe, and S. Karna. Cellular automata-based reconfigurable systems as a transitional approach to gigascale electronic architectures. *Journal of Spacecraft and Rockets*, 39(4):489–494, July 2000.

[LDK01] J. Lyke, G. Donohoe, and S. Karna. Reconfigurable cellular array architectures for molecular electronics. Technical Report AFRL-VS-TR-2001-1039, AFRL, March 2001.

[Lil30] Lilienfeld. Method and apparatus for controlling electric currents. U.S. Patent 1,745,175, 1930.

[Man95] R. T. Maniwa. Global distribution: Clocks and power. *Integrated System Design Magazine*, 1995.

[MDR+99] B. Martin *et al.* Orthogonal self-assembly on colloidal gold-platinum nanorods. *Advanced Materials*, 11:1021–25, 1999.

[Met99] R. Metzger. Electrical rectification by a molecule: The advent of unimolecular electronic devices. *Acc. Chem. Res.*, 32:950–7, 1999.

[MG02] M. Mishra and S. C. Goldstein. Scalable defect tolerance for molecular electronics. In *First Workshop on Non-Silicon Computing*, Cambridge, MA, February 2002.

[MH86] S. McFarling and J. Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, June 1986.

[Mir00] C. Mirkin. Programming the Assembly of Two- and Three-Dimensional Architectures with DNA and Nanoscale Inorganic Building Blocks. *Inorg. Chem.*, 39:2258–72, 2000.

[MMDS97] N. S. M.V. Martinez-Diaz and J. Stoddart. The self-assembly of a switchable [2]rotaxane. *Ang. Chem. Intl. Ed. Eng.*, 36:1904, 1997.

[MMK+02] J. K. N. Mbindyo *et al.* Template synthesis of metal nanowires containing monolayer molecular junctions. *J. Am. Chem. Soc.*, 124:4020–4026, 2002.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.

[MRM+00] J. Mbindyo *et al.* DNA-Directed Assembly of Gold Nanowires on Complementary Surfaces. *Advanced Materials*, 13(4):249–254, 2000.

[MSS+99] R. Mathews *et al.* A New RTD-FET Logic Family. *Proceedings of the IEEE*, 87(4):596–605, 1999.

[MW00] Merriam-Webster. *Webster's Third New International Dictionary*. Merriam-Webster, 2000.

[NF02] D. Nackashi and P. Franzon. Moletronics: A circuit design perspective. In *Proceedings of the SPIE*, volume 4236, 2002.

[NK01] M. T. Niemier and P. M. Kogge. Exploring and exploiting wire-level pipelining in emerging technologies. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 166–177. ACM Press, 2001.

[NPRG+02] S. R. Nicewarner-Pena, S. Raina, G. P. Goodrich, N. V. Fedoroff, and C. Keating. Hybridization and enzymatic extension of au nanoparticle-bound oligonucleotides. *J. Am. Chem. Soc.*, 124:7314–7323, 2002.

[NRK02] M. T. Niemier, A. F. Rodrigues, and P. M. Kogge. A potentially implementable FPGA for quantum dot cellular automata. In *Proceedings of the First Workshop on Non-Silicon Computation*, February 2002.

[Pee00] P. Peery. The drive to miniaturization. *Nature*, 406(6799):1023–1026, 2000.

[PG97] C. Pacha and K. Goser. Design of arithmetic circuits using resonant tunneling diodes and threshold logic. In *Proceedings of the 2nd Workshop on Innovative Circuits and Systems for Nanoelectronics*, pages 83–93, Delft, NL, September 1997.

[Pip90] N. Pippenger. Developments in "The Synthesis of Reliable Organisms from Unreliable Components". *Proceedings of Symposia in Pure Mathematics*, 50:311–324, 1990.

[PLP+99] H. Park, A. Lim, J. Park, A. Alivisatos, and P. McEuen. Fabrication of metallic electrodes with nanometer separation by electromigration. *Applied Physics Letters*, 75(2):301–303, 1999.

[Por98] W. Porod. Towards nanoelectronics: Possible CNN implementations using nanoelectronic devices. In *Fifth IEEE International Workshop on Cellular Neural Networks and their Applications*, pages 20–25, London, England, April 1998.

[PRS+01] D. J. Pena *et al.* Electrochemical synthesis of multi-material nanowires as building blocks for functional nanostructures. *MRS Symp. Proc.*, 636:D4.6.1–4.6.6, 2001.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In S. Verlag, editor, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture notes in computer science*, pages 151–166, 1998.

[PWW97] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[QPN+02] T. Quarles, D. Pederson, R. Newton, A. Sangiovanni-Vincentelli, and C. Wayne. The Spice page. http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/, 2002.

[RF93] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7(1):9–50, January 1993.

[RG02] D. Rosewater and S. Goldstein. Digital logic using molecular electronics. In *IEEE International Solid-State Circuits Conference*, February 2002.

[RKJ+00] T. Rueckes *et al.* Carbon nanotube-based nonvolatile random access memory for molecular computing. *Science*, 289(5476):94–97, July 7, 2000.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics (J. SIAM)*, 8(2):300–304, 1960.

[RWB+96] S. Roweis *et al.* A sticker based model for DNA computation. In *Proceedings of 2nd Annual DIMACS Workshop on DNA Based Computers (DIMACS)*, DIMACS, 1996.

[SBV92] J. R. S. Brown, R. Francis and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer, 1992.

[Sem97] Semiconductor Industry Association. *SIA Roadmap*, 1997.

[Semte] Sematech. *International Technology Roadmap for Semiconductors*, 2000 Update.

[SG59] M. Sobel and P. A. Groll. Group testing to eliminate efficiently all defectives in a binomial sample. *The Bell System Technical Journal*, 28:1179–1224, September 1959.

[Sha48] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.

[SKCA96] C. Stroud, S. Konala, P. Chen, and M. Abramovici. Built-in self-test for programmable logic blocks in FPGAs (finally, a free lunch: BIST without overhead!). In *Proceedings of IEEE VLSI Test Symposium*, pages 387–392, 1996.

[SKG00] S. K. Sinha, P. M. Kamarchik, and S. C. Goldstein. Tunable fault tolerance for runtime reconfigurable architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 185–192, 2000.

[SNJ+00] P. A. Smith *et al.* Electric-field assisted assembly and alignment of metallic nanowires. *Applied Physics Letters*, 77(9):1399–1401, August 2000.

[Spi96] D. Spielman. Highly Fault-Tolerant Parallel Computation. In *Proceedings of the 37th Annual IEEE Conference on Foundations of Computer Science 1996*, pages 154–163, 1996.

[SQM+99] Soh *et al.* Integrated nanotube circuits: Controlled growth and ohmic contacting of single-walled carbon nanotubes. *Applied Physics Letters*, 75(5), 1999.

[SS92] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. Digital Press, Bedford, MA, 1992.

[Sto] J. F. Stoddart. A [2]rotaxane ph controlled molecular switch. `http://www.chem.ucla.edu/dept/Faculty/stoddart/researc`

[SWHA98] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici. Built-in self-test of FPGA interconnect. In *Proceedings of IEEE International Test Conference*, pages 404–411, 1998.

[TDD+97] S. J. Tans *et al.* Individual single-wall carbon nanotubes as quantum wires. *Nature*, 386(6624):474–7, 1997.

[TEL95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, May 1995.

[TG99] R. R. Taylor and S. C. Goldstein. A high-performance flexible architecture for cryptography. In *1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 231–245, 1999.

[vECGS92] T. v. Eicken, D. E. Culler, S. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[vN45] J. v. Neumann. First draft of a report on the EDVAC. Contract No. W-670-ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia. Reprinted (in part) in Randell, Brian. 1982. Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin Heidelberg, June 1945.

[vN56] J. v. Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McCarthy, editors, *Automata Studies (Annals of Mathematical Studies)*, pages 43–98. Princeton University Press, 1956.

[VSS87] N. Viswanadham, V. Sarma, and M. Singh. *Reliability of Computer and Control Systems*, volume 8 of *North-Holland Systems and Control Series*. North Holland, 1987.

[WAM+02] S. Wind, J. Appenzeller, R. Martel, V. Derycke, and P. Avouris. Vertical scaling of carbon nanotube field-effect transistors using top gate electrodes. *Applied Physics Letters*, 80(20):3817–19, 2002.

[WB02] R. Weiss and S. Basu. The device physics of cellular logic gates. In *First Workshop on Non-Silicon Computing*, February 2002.

[WK01] S. Williams and P. Kuekes. Demultiplexer for a Molecular Wire Crossbar Network. U.S. Patent: 6,256,767, July 3, 2001.

[WKH01] S. Williams, P. Kuekes, and J. Heath. Molecular-Wire Crossbar Interconnect (MWCI) for Signal Routing and Communications. U.S. Patent 6,314,019, November 6, 2001., 2001.

[WLWS98] E. Winfree, F. Liu, L. Wenzler, and N. Seeman. Design and Self-Assembly of Two-Dimensional DNA Crystals. *Nature*, 394:539–544, 1998.

[Wol85] J. K. Wolf. Born again group testing: Multiaccess communications. *IEEE Transactions on Information Theory*, IT-31(2):185–191, March 1985.

[WT97] S.-J. Wang and T.-M. Tsai. Test and diagnosis of faulty logic blocks in FPGAs. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 722–7, 1997.

[Xil02] Xilinx Corporation. Virtex series fpgas. `http://www.xilinx.com/products/virtex.htm`, 2002.

[XRPW99] Y. Xia, J. Rogers, K. Paul, and G. Whitesides. Unconventional Methods for Fabricating and Patterning Nanostructures. *Chem. Rev.*, 99:823–1848, 1999.

[YMHB00] A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, June 2000.

[ZDRJI97] C. Zhou, M. R. Deshpande, M. A. Reed, and J. M. Jones II, L. anmd Tour. Nanoscale metal/self-assembled monolayer/metal heterostructures. *Appl. Phys. Lett*, 71:611, 1997.