NANA: A Nano-Scale Active Network Architecture

JAIDEV P. PATWARDHAN, CHRIS DWYER, ALVIN R. LEBECK,
and DANIEL J. SORIN
Duke University

This article explores the architectural challenges introduced by emerging bottom-up fabrication of nanoelectronic circuits. The specific nanotechnology we explore proposes patterned DNA nanostructures as a scaffold for the placement and interconnection of carbon nanotube or silicon nanorod FETs to create a limited size circuit (node). Three characteristics of this technology that significantly impact architecture are (1) limited node size, (2) random node interconnection, and (3) high defect rates. We present and evaluate an accumulator-based active network architecture that is compatible with any technology that presents these three challenges. This architecture represents an initial, unoptimized solution for understanding the implications of DNA-guide self-assembly.

Categories and Subject Descriptors: B.2.1 [**Arithmetic and Logic Structures**]: Design Styles; B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems); B.6.1 [**Logic Design**]: Design Styles; B.7.1 [**Integrated Circuits**]: Types and Design Styles; C.0 [**Computer Systems Organization**]: General; C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms: Design, Performance

Additional Key Words and Phrases: Accumulator ISA, active network, carbon nanotube, DNA, defect isolation, defect tolerance, nanocomputing, nanoelectronics, reverse path forwarding, self-assembly

1. INTRODUCTION

The semiconductor industry's roadmap identifies a "red brick wall" beyond which it is unknown how to extend the historical trend of ever-decreasing CMOS device size. "Eventually, toward the end of the Roadmap or beyond, scaling of MOSFETs will become ineffective and/or very costly, and advanced non-CMOS

This work is supported an NSF ITR grant CCR-0326157, a grant from Duke University Provost's Common Fund, an AFRL contract FA8750-05-2-0018, a Warren Faculty Scholarship (Sorin) and equipment donations from IBM and Intel. We thank the members of the TROIKA project and Lavanya Ramakrishnan for their help with this work.
Authors' addresses: J. P. Patwardhan, A. R. Lebeck, Department of Computer Science, Duke University, PO Box 90129, Durham, NC 27708-0129; C. Dwyer, D. J. Sorin, Department of Electrical and Computer Engineering, Duke University, Box 90291, Durham, NC 27708-0291; email: {jaidev, alvy}@cs.duke.edu, {dwyer,sorin}@ee.duke.edu.

solutions will need to be implemented" [International Technology Roadmap for Semiconductors, 2002 Update, Difficult Challenge #10].

Technology change is fuel for architectural innovation. Evolutionary changes in CMOS have inspired research on several important topics including wire dominated designs, power dissipation, and fault tolerance. A revolutionary technology change, such as replacing CMOS, is a potentially disruptive event in the design of computing systems.

Emerging technologies for further miniaturization have capabilities and limitations that can significantly influence computer architecture and require reexamining or rebuilding abstractions originally tailored for CMOS. This article explores the architectural challenges introduced by emerging bottom-up fabrication of nanoelectronic circuits and develops an architecture that meets these challenges.

We focus on one specific nanotechnology in this article: DNA-guided self-assembly [Seeman 1999] of carbon nanotube field effect transistors (CNFETs) [Bachtold et al. 2001; Fuhrer et al. 2001; Huang et al. 2001] and wires. To place and interconnect these components, we propose using patterned DNA nanostructures [Yan et al. 2003b] as a scaffold to which we attach carbon nanotubes. The DNA nanostructures create a limited size circuit (node) of CNFETs. DNA-guided self-assembly can also provide a scaffold for metal that forms the interconnect between nodes, but without the control available in the patterned nanostructures, thus producing a random interconnect. There are three aspects of this technology that significantly impact architecture: (1) limited node size, (2) random interconnection of nodes, and (3) high defect rates. Our goal is to develop an appropriate architecture that can be implemented in any technology with these characteristics. We also enumerate several important issues to address during architectural development.

There are likely many possible approaches to developing a functioning system. Our goal in this work is not to determine the best approach, rather it is to simply obtain one approach. Therefore, in this article we adopt the philosophy of "make it work first, optimize later." We present one potential solution: an active network architecture with an accumulator-based ISA. The limited node size prevents the design of a single node that can perform all operations. Instead, we design different node types (e.g., add, memory, shift) based on node size constraints. A configuration phase at system startup maps out defective nodes and links, organizes a memory system, and sets up routing in the network. To execute, an instruction searches for a node with the appropriate functionality (e.g., add), performs its operation, and passes its result to the next dependent instruction. In this active network execution model, the accumulator and all operands are stored within a packet rather than at specific nodes, thus reducing per-node resource demands. The active network execution model enables us to encode a series of dependent instructions within a single packet.

This architecture matches our technology characteristics since it: (1) allows for differing node types with specialized functionality, (2) tolerates a random interconnection of nodes, and (3) tolerates node and interconnect fabrication defects. While the architecture has limitations, our design demonstrates that it is possible to build a general purpose computing system using self-assembled

nanoelectronic devices despite severe technological constraints. As a first step, the nano-scale active network architecture (NANA) does remarkably well and provides valuable lessons for future designs. We believe that NANA is a necessary first step toward exploiting nanotechnology's potential to overcome the "red brick wall." The contributions of this article are:

—We present a list of challenges that are likely to be encountered by system architects when building a system using self-assembled networks of simple computational circuits.
—We adapt an existing algorithm to provide defect isolation for node defect rates up to 30%.
—We propose and evaluate a general purpose architecture built using self-assembled networks of simple computational blocks, demonstrating that we can build a computing system despite the hurdles presented by the underlying technology.
—We identify key aspects of the architecture that need to be improved further to achieve better performance.

The rest of this paper is organized as follows. Section 2 describes DNA-guided self-assembly of nanoelectronic components and Section 3 discusses the architectural implications of this technology. We describe our proposed architecture in detail in Section 4 and present an evaluation of the architecture using two illustrative examples in Section 5. Section 6 discusses related work and Section 7 concludes.

2. EMERGING NANOTECHNOLOGIES

In this section, we describe the specific nanotechnologies used in this article. We discuss the electronic components (Section 2.1), DNA self-assembly of these components into circuit nodes (Section 2.2), and the large-scale interconnection of these circuit nodes (Section 2.3).

2.1 Carbon Nanotube Electronics

There are many choices for constructing nanoelectronic devices and nanowires [Bachtold et al. 2001; Cui and Lieber 2001; Huang et al. 2001; Martin et al. 1999; Tans et al. 1998; Tour 2000]. One such promising nanoelectronic device is a carbon nanotube field effect transistor (CNFET) [Fuhrer et al. 2001; Javey et al. 2004; Kim et al. 2004; Tans et al. 1998; Wind et al. 2002], in which application of a gate voltage modulates the conductivity of a semiconducting nanotube. Recent advances enable the separation of metallic nanotubes from semiconducting nanotubes, precisely controlling the length of individual nanotubes [Strano et al. 2003, Zheng et al. 2003] and self-assembly of carbon nanotube based electronic devices [Hazani et al. 2004]. Therefore, we could use both types of carbon nanotubes to construct logic gates, memory (e.g., with cross-coupled NOR gates), and circuit interconnect. Other potential materials (e.g., nanorods [Martin et al. 1999], silicon nanowires [Cui and Lieber 2001; Huang et al. 2001]) could be substituted for the carbon nanotubes without loss of generality in our architectural analysis.

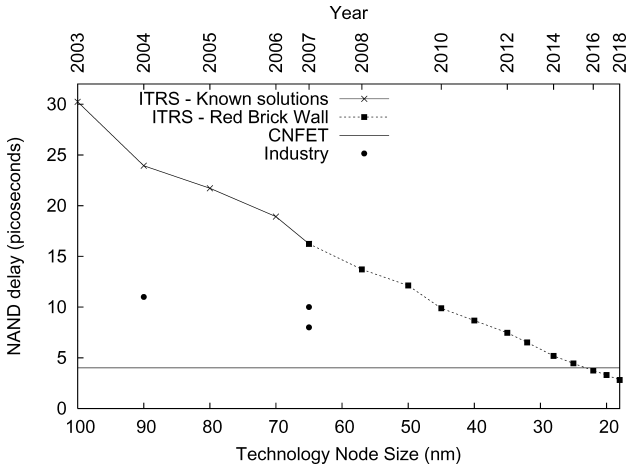


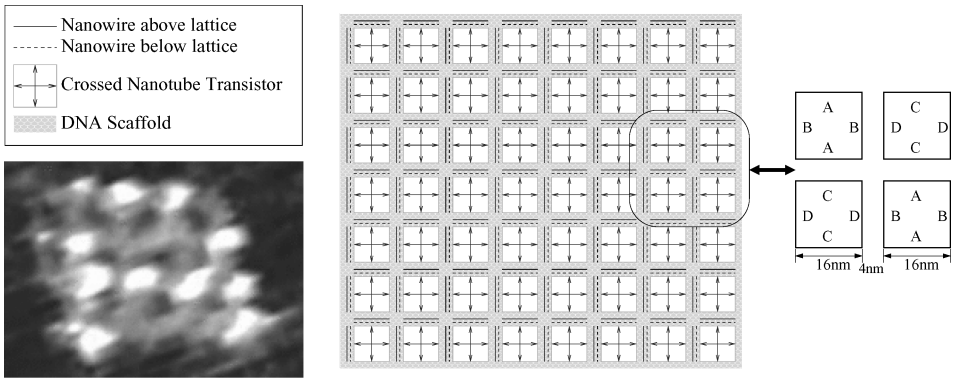
Fig. 1. Nanoscale device performance.

To explore the potential of CNFETs, we simulate several circuits using a customized SPICE 3f5 kernel that models CNFET behavior in logic gates [Dwyer et al. 2004b]. We compare CNFET-based logic gates with CMOS using ITRS target values and some data from industry processes. Figure 1 shows a NAND gate delay for each approach. To obtain these values we load each circuit output with four inverters (FO-4) and pass a square input signal through a series of four inverters to each circuit input. We derive the CNFET I/V behavior, parasitic capacitances, and inductances from geometric and literature values [Burke 2003; McEuen et al. 2002]. Our results indicate that CNFET circuit performance is deep within the "red brick wall" predicted by the ITRS. Industry data shows much better performance for CMOS NAND gates, but the improvements across process generations is slowing down. The CNFET results are also pessimistic, as the theoretical limit is significantly higher [Dürkop et al. 2004]. The added benefit that CNFETs are amenable to self-assembly makes this an attractive alternative or supplement to silicon device technology.

2.2 DNA Tiles and Nanostructures

The precise placement and interconnection of individual carbon nanotubes remains an area of diverse research. These integration challenges and their impact on higher-level designs are shared by other emerging technologies (e.g., silicon nanowires, quantum dots, etc.). Since these devices are smaller than the resolution of top-down photolithographic methods, research has explored various techniques for bottom-up self-assembly.

To overcome the challenge of nanoelectronic integration, we propose using DNA self-assemblies to produce patterned nanostructures onto which we can programmably attach carbon nanotubes. DNA's well-known double-helix structure is formed through its well-understood base-pairing rules—adenine (A) to thymine (T) and cytosine (C) to guanine (G). By specifying a particular sequence of base pairs on a single strand of DNA, we can exploit the base-pair rules as organizational instructions [Seeman 1999].



a) Patterned letter 'A' on DNA-lattice

b) Proposed patterning method for DNA-lattice and carbon-nanotube self-assembly

Fig. 2. A DNA scaffolding for carbon nanotube circuits.

These DNA tags can be used to create 2D-patterned nanostructures [Winfree et al. 1998]. For this article we focus on a particular structure that creates a 'waffle'-like lattice with repeating cavities of ~16 nm × 16 nm and 4 nm separation between cavities [Yan et al. 2003b]. This type of lattice has been experimentally demonstrated and can achieve sizes that extend to 3 μm on each side (i.e., more than 150 cavities on a side).

Recently, we demonstrated the ability to place aperiodic patterns on a smaller lattice [Park et al. 2006], which could enable the placement of carbon nanotubes or nanowire transistors [Skinner et al. 2005] at arbitrary locations in the lattice. Figure 2(a) shows an atomic force microscopy (AFM) image of an 80 nm × 80 nm lattice with the letter 'A' patterned on it. We can place and interconnect carbon nanotubes by forming tags (Figure 2(b)) at specific points on the lattice [Dwyer et al. 2005; Yan et al. 2003a] and using a recently demonstrated technique for attaching the appropriate complementary DNA tags to carbon nanotubes [Dwyer et al. 2002]. Connections between nanotubes are formed using a technique called electroless plating [Braun et al. 1998].

The technologies described in this section provide a set of potential building blocks for constructing nanoscale systems, and more details are provided elsewhere [Patwardhan et al. 2004]. The demonstrated operation of CNFETs and the ability to attach DNA tags to them make this a promising nanoelectronic technology. The DNA self-assembly technique is independent of the specific nanoelectronic device used, however, the limited size of each lattice (node) presents challenges for creating large sophisticated circuitry. We now discuss how to interconnect these nodes into a computational substrate.

2.3 Large-Scale Interconnection

Using inexpensive laboratory equipment we could potentially use self-assembly to simultaneously build as many as 10^{12} identical, but small, nodes. This number of nodes, if placed 0.25 μm apart, would cover a 325 cm × 325 cm area,

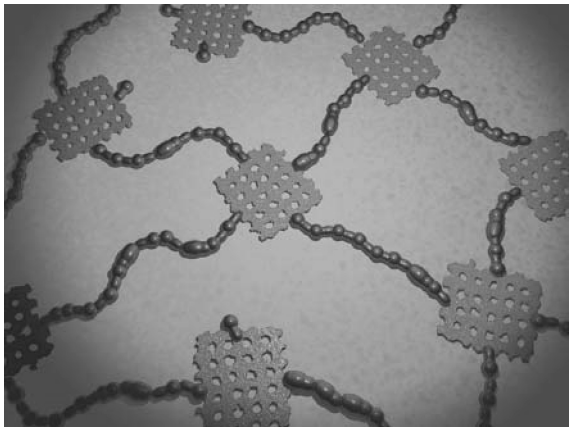


Fig. 3. Schematic rendering of a self-assembled DNA interconnection network after metal decomposition.

or the equivalent area of ∼150 wafers (300 mm diameter). Although the size of an individual node is well above the minimum feature size of photolithography, the number of nodes fabricated through self-assembly limits how heavily the overall process can rely on conventional patterning. Self-assembling nodes onto a substrate at well-defined places is also difficult without "naming" each placement site (pick-and-place methods will not scale to this number of components). Even with DNA tags on the substrate, the nodes are not guaranteed to fall into place precisely. Most conventional architectures require precise placement and interconnection between circuits. Therefore, even if we could use a conventional photolithographically patterned network to interconnect nodes, the result would be a random interconnection due to the random placement of nodes on the substrate. This is the sacrifice a self-assembly process imposes: precision and control exist only at small length scales (∼2 μm, for now).

We use individual DNA strands that self-assemble between node edges, providing a scaffold for metal that forms an electrical connection [Liu et al. 2004; Yan et al. 2003b]. This larger scale process cannot deliver the precise control found in the earlier process used to assemble the nodes, but it can fabricate single wire interconnections between the edges of the nodes, as illustrated in Figure 3. In this article, to simplify presentation, we model system fabrication using a uniform grid and introduce defective nodes and links. Furthermore, preliminary evaluations comparing the grid approach to a physical model (based on a random walk) of DNA self-assembly of interconnections reveals that the two techniques produce similar overall network characteristics.

3. ARCHITECTURAL IMPLICATIONS

The DNA-guided self-assembly process described in Section 2 presents several challenges that must be addressed when designing a system. The three primary aspects of the fabrication process are small-scale control of placement and connectivity within a single node (Section 3.1), large-scale randomness in node placement and interconnection (Section 3.2), and high defect rate (Section 3.3).

These three aspects significantly impact architectural decisions (Section 3.4), particularly since conventional architectures assume precise control at both the small-and large-scale.

3.1 Small-Scale Control

The ability of DNA-guided self-assembly to achieve only small-scale control impacts architectural decisions in several ways. Three of the most significant are: limited space, limited coordination, and limited communication.

Limited space. A 150 $\times$ 150 node can have a maximum of 22,500 CNFETs, however, on-node interconnect will reduce efficiency since a node only has two levels of interconnect. Furthermore, a portion of each node must be allocated as a "pad" for the DNA interconnect to other nodes.

The limited node size presents a trade-off in node design. At one extreme, we could design just a single node type that contains both computation and storage capabilities. However, since storage and computation circuits must share the node, each may be severely limited in capability. Alternatively, we could design a few specialized node types, some devoted to computation and others to storage. Even when designing a specialized node, the limited space impacts architectural decisions. For example, large state machines are not an option within a node since there is insufficient space for state storage. Similarly, the number of bits available in a storage node may be limited, thus affecting an architecture's word size.

Limited communication. Without large-scale control, there is limited communication among nodes. Each node has four neighbors and there is no long haul communication. Furthermore, the connections between nodes are limited to single wires. Although the degree of each node or the number of connections between neighbors could be increased, each connection occupies precious edge space. By contrast, conventional CMOS designs exploit multiple metal layers for long-haul communication and large-scale control to create multi-wire connections between components.

Limited coordination. Conventional CMOS designs rely on precise control during fabrication to create sophisticated circuits (e.g., 64-bit adder with carry lookahead). For our technology, if the most sophisticated node is a full-adder, then it is unlikely that 64 such nodes can be coordinated to implement a 64-bit adder. Coordination among nodes is limited to immediate neighbors and it is difficult *a priori* to configure a group of nodes to operate in a coordinated manner.

3.2 Large-Scale Randomness

Our proposed self-assembly process provides excellent control at the small-scale, however, it cannot achieve such control at large scales. The resulting randomness introduces some additional issues that architectures must address.

Random node placement. The self-assembly process does not guarantee where any particular node will lie in the final circuit. Each node simply attempts to connect to other nearby nodes. The architecture and machine organization must accommodate this arbitrary placement of functional blocks.

Random node orientation. Similar to the random node placement, the assembly process we envision does not provide control over node orientation. Any system design must tolerate arbitrary node orientations and cannot make *a priori* assumptions on orientation. For example, it is incorrect to assume that the "east" side of one node will connect to the "west" side of its adjacent node.

Random node connectivity. Connections between nodes are not guaranteed to succeed during self-assembly. Therefore, it is possible for any node to have between zero and four functioning connections to its neighbors. The architecture must not make any *a priori* assumptions about available connectivity. When combined with random orientation, it is possible for nodes to connect in a triangular shape rather than the 2×2 grid one would assume with nodes that have a degree of four.

3.3 High Defect Rates

An inherent aspect of any self-assembly process is defects. Fabrication defects can influence node functionality and connectivity. Some interconnect defects cause the above problems with connectivity. While some aspects of fabrication can reduce the likelihood of defects (e.g., purification steps or overdesign of DNA tags), there will always be a significant number of defects and any architecture using these technologies must tolerate them.

3.4 Architectural Challenges

The above discussion exposes several aspects of this fabrication technique for nano-scale circuits that must be addressed by any architecture and its corresponding implementations. In this subsection, we enumerate several important challenges to developing an appropriate architecture for this emerging technology. This list is not exhaustive, but rather highlights some important challenges.

Designing nodes. The architect must decide what functionality to place in each node. Should there be homogeneous or heterogeneous nodes? If heterogeneous, then what types of nodes? How does node design affect connectivity/communication with other nodes, and what primitives should be provided?

Utilizing multiple nodes. Since individual nodes do not contain sufficient computation and storage to perform much useful work in isolation, an architect must determine how to exploit multiple nodes. This must be achieved given the above limitations on coordination, communication, placement, orientation, and connectivity.

Routing with limited connectivity. Traditional routing techniques may not apply, since there is limited space for the complexity of dynamic routing and there are insufficient guarantees on node placement and connectivity to use conventional static routing.

Developing an execution model. The execution model embodies the software-visible aspects of the architecture and can be influenced by implementation constraints or instruction set requirements. For the envisioned fabrication technique, the execution model must overcome the severe implementation constraints outlined above while enabling a reasonable instruction set.

Developing an instruction set. Programmable systems require an interface that enables software to specify operations. Typically, this is achieved by the instruction set architecture (ISA). The ISA may be influenced by the underlying capabilities of the technology. Given our fabrication technique, the architect must design an appropriate ISA that supports the above execution model.

Providing a memory system. Storage is a crucial component of most computing systems, regardless of the execution model. The ability to store values for future use and to name and find particular values is a necessary aspect of most computing paradigms.

Interfacing to the micro-scale. An important aspect of any nano-scale system is the interconnection to larger-scale components (e.g., micro-scale). This connection is necessary for (at least) providing an I/O interface for communication with the outside world. It may be possible for the architecture to exploit this interface in other ways.

The challenge is to address each of these issues such that we arrive at a functioning system. There are likely many possible approaches to developing a functioning system. Our goal in this work is not to determine the best approach, rather, it is to simply obtain one approach. With any emerging technology, we must limit the scope of studies to ensure forward progress. The remainder of this article presents one potential architecture.

4. AN ARCHITECTURE FOR SELF-ASSEMBLED NANO-ELECTRONICS

As an initial approach to address the issues raised in Section 3, we propose NANA, an active network architecture that is compatible with our fabrication technology. The architecture is like an active network [Tennenhouse and Wetheral 1996] in that execution packets that contain instructions and operands search through a loosely-structured sea of processing and memory nodes for the functionality that they need at each step of execution. This architecture matches our technology characteristics since it (1) allows for differing node types with specialized functionality, (2) tolerates a random interconnection of nodes, and (3) tolerates node and interconnect fabrication defects.

4.1 System Model

The system model is a random interconnection of various node types in which all nodes contain circuitry for communication and each node has some specialized circuitry (e.g., processing, memory, etc.). Groups of nodes are organized into cells. A node communicates with a neighboring node via a single link that is asynchronous and bidirectional (time-multiplexed on a single physical wire). Each cell has a via that is its connection to the micro-scale, and one of the nodes connected to the via acts as the anchor node for the cell. Inter-cell communication occurs through a micro-scale interconnection network. The memory nodes in each cell comprise a portion of the global memory space. Some fraction of nodes are configured as memory ports to provide an interface between execution packets and memory storage. Figure 4 illustrates our system model. To impose structure on the interconnection network and the memory system, there is a configuration phase [Patwardhan et al. 2005] that occurs before any execution.

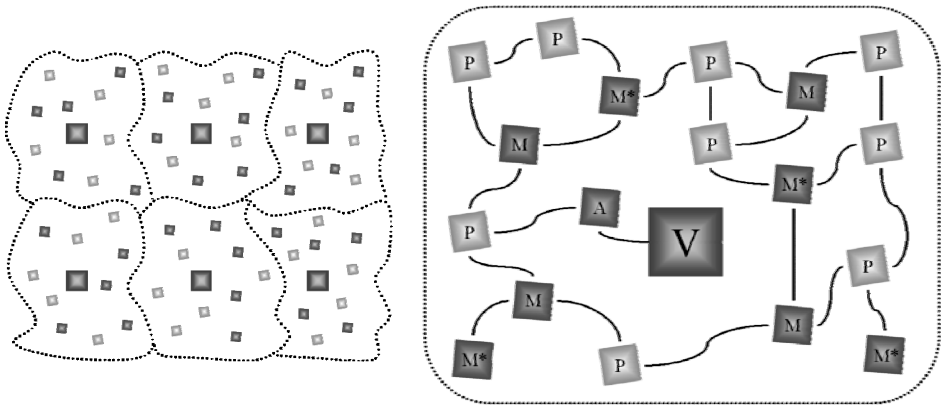


Fig. 4. (a) System model. (b) Processing nodes (P), memory nodes (M), memory port nodes (M*), anchor node (A), and via (V). This schematic is not to scale (w.r.t. nodes per cell).

Reconfigurable architectures [Culbertson et al. 1996; DeHon 2002; Goldstein and Budiu 2001; Heath et al. 1998] have demonstrated that this approach is important in order to achieve high performance in the context of highly-focused (i.e., aggressive) or highly-defective technologies, including nanotechnology. We describe the purpose, beyond defect tolerance, and operation of the configuration in detail later in this section.

While node functionality is heterogeneous, all nodes have some common responsibilities. Each node generates its own local clock (we choose a clock frequency of 10 GHz, which is pessimistic, given the data in Figure 1) and communicates asynchronously with its neighboring nodes using signaling techniques similar to push-style pipeline systems. High-level communication between two devices over a single wire can be managed using simple two- and four-phase single wire techniques [van Berkel 1996]. Each node must also contain routing functionality for determining the outgoing link for an incoming packet (or the result of an operation). This circuitry maintains node state (e.g., currently processing a packet) and handles link contention.

4.2 Execution Model

The execution model relies on an accumulator-based ISA. Conceptually, the accumulator is initialized and then a sequence of operations is performed on the corresponding series of operands. The accumulator-based ISA reduces the need for widespread *a priori* coordination and communication among many components (e.g., associative lookup in issue queues), since the only data dependence involves the accumulator, and instructions are processed in order [Kim and Smith 2002]. We support accumulator-based execution by forming an execution packet that contains the operations, the accumulator, and all operands in appropriate order. Instructions are executed in the order specified in the packet, as they are routed through the network and find the requisite functional units (or memory ports). Logically, each functional unit performs its specified operation,

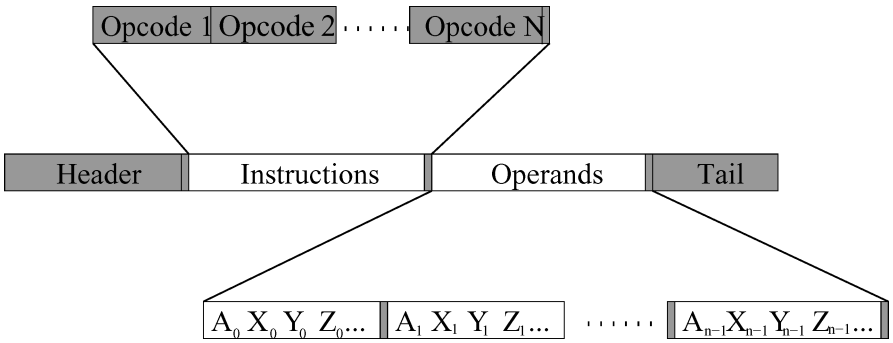


Fig. 5. Packet format.

removes the operand and forwards the new accumulator and the remaining operands to the subsequent functional units. Each subsequent functional unit performs a similar sequence until all operations in the packet are completed. Memory operations generate memory packets that are handled by the memory ports, as discussed in Section 4.5. Packet sequencing is achieved using a process called chaining, discussed in Section 4.6.

Our system and execution model enables significant parallelism by instantiating multiple execution packets within a cell and in multiple cells. While this parallelism is an important aspect of our architecture that fully exploits the capabilities of the underlying technology, in this article we focus primarily on the operation of a single cell and sequentially instantiate execution packets.

To augment the defect tolerance of configuration and to protect against transient faults, we could add a signature vector to each packet and verify the integrity of a computation performed by the packet. The signature vector is operated on like the operands field of a regular execution packet, with the exception that the initial signature is not consumed by the operation. The order of instructions will be reflected by a characteristic signature vector and can be used to determine if the nodes performing those operations were functioning properly during the signature calculation. This approach can be further augmented with redundant execution packets and a voting mechanism.

4.3 Instruction Set and Packet Formats

The format of an execution packet is: header, instructions, operands, tail. Specific bit patterns delineate field boundaries. The header is a fixed-length field that includes packet type and other metadata. The instructions field is a variable-length list of opcodes in program order. The operands field is a variable-length list of operand values. To accommodate the limited node size, we use a bit-serial implementation. The active network architecture and accumulator ISA are independent of this choice and provide an architecture that can scale with improvements in node capabilities (i.e., multi-bit operations). Figure 5 shows the execution packet format for our bit-serial implementation. The operands field is divided into bit-slices from least significant bit to most significant bit (from packet head to tail). Each bit-slice starts with a bit from

Table I. NANA Instruction Set

Instruction Type	Instructions
Arithmetic	ADD, INC, SUB, DEC, SHL, SHR
Comparison	COMPEQ, COMPGT, COMPLT, SETEQ, SETGT, SETLT, SETZ
Operand Stream Control	LDCONST0, LDCONST1, CPACC, MOV, DELOP, OPFLUSH, SWAP
Logical	AND, NAND, NOR, NOT, OR, XOR, XNOR, NOP
Load	LD [Mem], LDI [Mem]
Store	ST [Mem], STI [Mem]
Conditional Store	CST [Mem], CST_RST [Mem], CRST [Mem], CSTI [Mem], CSTI_RST [Mem], CRSTI [Mem]
Unconditional Control Transfer	JMP [Mem], CALL [Mem],JMPI [Mem],CALLI [Mem]
Conditional Control Transfer	CALLNZ [Mem], CALLZ [Mem], CALLNZI [Mem], CALLZI [Mem]

the accumulator and is followed by each bit (for the particular bit-slice) of the operands.

The instructions that this architecture supports must be bit-serial in nature and require little communication between bit-slices. Many instructions are simple to implement with limited circuitry (e.g., ADD, SUB, OR, AND, XOR, NOR, NAND, compare, move) and require only small extensions to a bit-serial full adder circuit. Each operation requires only a small amount of information (e.g., carry-out bit) to be communicated to subsequent bit-slices. This simplifies the implementation details of the circuits so that they will fit within the node size limits of the technology. Although each instruction is bit-serial, the bit interleaving enables parallel execution of consecutive operations in a pipelined manner. Instructions supported by NANA can be divided into nine categories and are listed in Table I.

The serial nature of this architecture and the limited node complexity of the technology make certain operations difficult. Table II lists several instructions specially designed to help overcome these difficulties. For example, right shifts (moving bits from the tail toward the head) are difficult because they require bits to be forwarded ahead of other bits unless entire operands are stored at the functional node. Since we assume that both operand storage and ALU-type functionality in a single node requires too much area for our limited node size, we exploit the stack-like nature of the operand stream to support right shifts. When a right shift is executed, it also places the result at the end of the operand stream. Thus, to execute a right shift, we buffer the field separator between bit-slices and send out the next observed data bit before re-inserting the field separator into the packet bit stream.

The bit-slice packet encoding also complicates memory operations. For example, a load requires all of its address bits to generate a request. If the address is in the operand stream, then it is impossible for the load to interleave the resulting data in the same operand stream, since all the low-order bits are ahead in the packet flow before the entire address is obtained. Similar difficulties exist for stores. Therefore, a packet cannot both calculate an address and use it in the same packet. To address these limitations, we provide three

Table II. Definitions of a Selected Subset of Instructions

Instruction	Operation
MOV	Move accumulator to end of operand stream
SWAP	Swap first and second operand
SHR	Shift accumulator right by 1 bit, move accumulator to end of operand stream
DELOP/OPFLUSH	Remove one/all operands from operand stream
CPACC	Create copy of accumulator at end of operand stream
SET (EQ/GT/LT/Z)	Set flag bit in tail if condition satisfied, consume accumulator
COMP (EQ/GT/LT)	Set flag bit in tail if condition satisfied, consume first two operands
LDI [Mem]/ STI [Mem]	Load/store indirect through constant address [Mem]
CST [Mem]/CSTI [Mem]	Conditional store direct (CST) or indirect (CSTI) to [Mem] (status bit in tail must be set)
CST_RST [Mem]	Conditional store to [Mem], reset status bit after performing store
JMP [Mem]/JMPI [Mem]	Fetch instructions into existing packet from direct (JMP) or indirect (JMPI) address [Mem]
CALL [Mem]/CALLI [Mem]	Create new packet using instructions from direct (CALL) or indirect (CALLI) address [Mem]
CALLNZ [Mem]/CALLNZI [Mem]	Fetch instructions into new packet if status bit is set (not zero) (direct/indirect)
CALLZ [Mem]/CALLZI [Mem]	Fetch instructions into new packet if status bit is not set (zero) (direct/indirect)

specific types of memory addressing: immediate, constant address and indirect address. Constant addressing requires the address to appear in the instruction field of the packet. Indirect addressing supports indirection through a memory location that is specified as a constant in the instruction field of the packet. We also provide special instructions (JMP & CALL) for instruction sequencing (discussed in Section 4.6). Conditional execution is supported through status bits (e.g., condition codes) in the packet tail. Currently, we support conditional store and CALL instructions that must wait to execute until the packet tail arrives so that they can examine the appropriate status bit.

Programming NANA is similar to programming other accumulator based ISAs [Campbell-Kelly 1998; Kim and Smith 2002, 2003; Lavington 1978], however, care must be taken to account for system capabilities and constraints. For example, the 'shift right' instruction (SHR) is constrained by node resources to shift the accumulator and move it to the end of the operand stream, while the 'shift left' instruction (SHL) operates as expected (i.e., it shifts the accumulator left by one bit). Another constraint arises from the structure of the memory system—all loads must precede stores in a packet. Consider a simple code fragment ($x = x + *(y + a)$) that computes a memory address ($y + a$) and then adds the contents of that location to another variable stored in memory. Due to the load-store ordering constraint, instructions must be divided into two packets. Table III shows the two packets needed to implement the code segment, and how their fragments are arranged in memory. The first packet, starting at address 0×10, performs an address calculation ($y + a$) and stores the result in a

Table III. Memory Layout for Two Packets that Compute x=x+*(y+a)

Address	Instruction	NextPC		Address	Instruction	NextPC
0×10	LD y	0×14		0×40	LDx	0×44
0×14	LD a	0×18		0×44	LDI z	0×48
0×18	ADD	$0 \times 1A$		0×48	ADD	$0 \times 4A$
$0 \times 1A$	ST z	$0 \times 1E$		$0 \times 4A$	STx	0×0
$0 \times 1E$	CALL(0×40)	0×0				

third location, z. The last instruction, at address 0×20, chains this packet to the next packet, which starts at address 0×40. The second packet performs the addition of x with the value stored at the memory location pointed to by z, and stores the result into x (i.e., $x = x + *z$). This packet executes by first loading the value of x, then performing an indirect load on z (instruction at 0×44). Next, it executes the add and stores the result into x. This example illustrates some constraints that must be faced in programming NANA. We expect that as the underlying technology matures, a richer ISA with more complex instructions will become possible, including efficient variable bit shifts, bit-serial multiplication, and division. Until then, we compose these more sophisticated operations in software using simpler primitives.

4.4 Interconnection Network: Finding Resources for Execution

The active network architecture must enable packets to find what they need without deadlocking or livelocking, despite high defect rates and traveling through a randomly-interconnected sea of nodes. To avoid request/response deadlock (i.e., fetch deadlock), the minimum requirement is three logical networks: one for execution packets, one for memory request packets, and one for memory response packets. Each of these logical networks is irregular and must provide deadlock- and livelock-free routing. While we could implement these three networks using three virtual channels [Dally 1992] per unidirectional link, this unnecessarily increases the amount of buffering required on a single node. We reduce the requirement to two virtual channels per unidirectional link by creating distinct physical networks for execution and memory; we explain how this is implemented in Section 4.5. We also use wormhole routing since it requires the least buffering on each node (1 bit per channel).

4.4.1 *Imposing Structure with Gradients.* Virtual networks avoid fetch deadlock, yet each network must still provide deadlock- and livelock-free routing. Given our irregular networks, we create a spanning tree using the reverse path forwarding algorithm (RPF) [Dalal and Metcalfe 1978; Patwardhan et al. 2005] and then employ a variant of up*/down* routing [Schroeder et al. 1991], a degenerate case of turn-model routing [Glass and Ni 1992], and back pressure flow control. The challenge is implementing these techniques with limited node functionality.

To meet this challenge, we equip each node with two forms of communication: (1) broadcast and, (2) routing along gradients [Johnson and Maltz 1996; Intanagonwiwat et al. 2000]. Packet headers include information on the type of

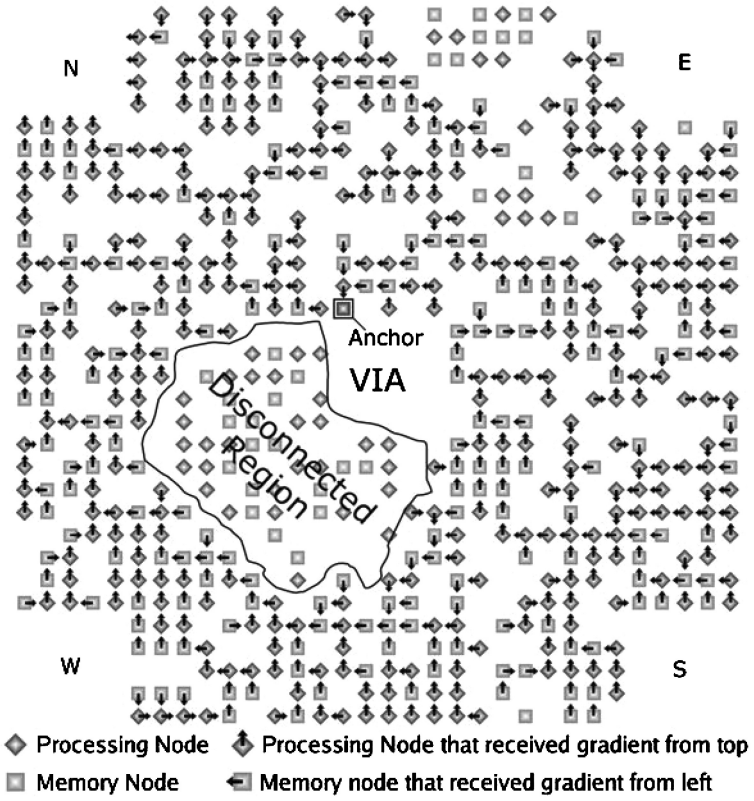


Fig. 6. A 32×32 grid of memory and processing nodes with one established gradient (North).

communication to use. Broadcast requires minimal state per node and is used during configuration only. Gradients reduce per-node resources while still enabling deadlock- and livelock-free routing. We use the RPF algorithm to create a spanning tree with a specific via as the root and establish a gradient with a specialized packet. Each node marks the link on which the gradient packet was received (i.e., points to its parent in the spanning tree) and broadcasts the packet to its other neighbors. A node will not broadcast gradient packets after having seen the first packet. This process can be generalized to any number of gradients if each node records an identifier for each gradient it detects. The broadcast algorithm terminates when all reachable nodes have received the gradient. There is no external action required to terminate the algorithm, and each node automatically stops forwarding broadcast packets when it has been configured.

We use five gradients: one for each planar direction (north, south, east, and west), and an additional gradient that establishes cell boundaries and the direction toward the via in each cell (called the cell gradient). The planar gradients are established by starting the broadcast at the north, south, east, and west edges (or corners) of the system, respectively. Figure 6 illustrates a gradient established from the upper left corner (north) in a 32×32 grid with a 30% defect

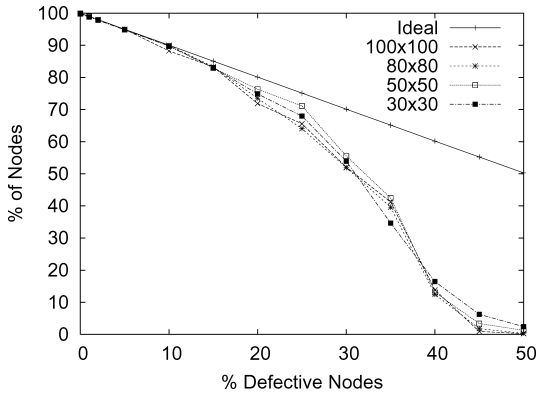


Fig. 7. Percentage of nodes reachable by gradient broadcast for varying defect rates.

rate. Defective nodes, not drawn in this figure, can cause islands of disconnected nodes such as the region near the via.

Configuring Cells. The process is initiated at each via in parallel by broadcasting a cell ownership packet that includes a cell identifier. The cell gradient broadcast stops when its wave front collides with the wave front from an adjacent via. Nodes detecting a conflict in cell identifiers stop the broadcast, creating a boundary between cells, and record that they are boundary nodes.

Tolerating Defects. Creating spanning trees using a broadcast flood maps out defective nodes and links, since no other node will have a gradient pointer to the defective node. If routing is restricted to follow a gradient, then packets will never be sent to a node that did not receive a gradient packet. Figure 7 shows the percentage of nodes that can be reached by a broadcast for increasing node defect rates. Each point on a curve is the average of ten simulation runs with different distributions of defects in the network. The different curves correspond to different network sizes. For defect rates up to 20%, the broadcast reaches most of the functional nodes. A majority of functional nodes is still reachable for defect rates up to 30%, beyond which we see a sharp drop in the number of nodes receiving the broadcast because increasingly large regions are isolated from vias. Our analysis also shows that it is better to broadcast the planar gradients from an edge than from a corner of a rectangular or square network of nodes. In general, a via with more nodes surrounding it has a better chance of reaching a larger set of nodes.

Our defect model assumes that the routing circuitry for a node is either fully functional or not operational at all.[1] We can tolerate shorts in the node interconnect, and we call such defects broadcast defects because they represent the unintentional broadcast (to more than one link) of packet bits. Such defects are difficult to avoid during fabrication and require an arbitration scheme, similar to fixed back-off media access schemes in networks. The asynchronous link controllers in each node can be designed to assert a link-good signal after

[1]The general Byzantine defect model, in which defective nodes can produce arbitrary behavior, has been considered in the internet literature, but tolerating such defects requires a great deal of complexity at each node [Castro and Liskov 1999].

a random interval of time after power-up. The randomness can be introduced during the self-assembling process [Dwyer 2003]. Every node monitors its links for the link-good signal and marks any link that has received more than one signal as defective. When the node's internal random interval has elapsed, if the link is not already marked defective it asserts its own link-good signal on all links. This arbitration scheme identifies both shorts and opens on links between nodes. The nodes connected to the via essentially share a single link (the via) that appears as a broadcast defect. The result of this arbitration scheme is for a single node to remain actively connected to the via, thus acting as the cell anchor.

Due to defects, some vias may not have a path to any of the four planar gradient destinations. This can be detected by monitoring the via at the micro-scale during the broadcast of each of the planar gradients. If the via fails to receive any of the gradient broadcast packets, it should be marked as defective and not participate in cell configuration.

4.4.2 *Execution Packet Routing.* The spanning tree structure imposed by gradients provides the framework for packet routing. Execution packets and memory packets never share physical links and thus cannot block each other. Up*/down* routing on the spanning trees prevents routing deadlock and livelock. However, execution packets must be able to find the necessary resources for execution, and memory packets must successfully find the appropriate memory location, which responds if necessary. We discuss memory packet routing in Section 4.5.

To avoid deadlocking execution packets, we simply follow a single gradient (up* on one spanning tree) on one virtual channel until we reach a cell boundary, then reflect the packet back into the cell on the opposite planar gradient but on the other virtual channel. Reflection only occurs if there are remaining instructions in the packet, otherwise a special packet is sent to the anchor node to indicate completion. We note that the header can run ahead of the operand stream allocating nodes for instructions (due to execution delay in a node). This approach can indefinitely bounce a packet between cell edges. The only constraint is that packet length must be less than the total number of nodes in the round trip traversal. Since execution packets only traverse in the up* direction of the spanning tree, each node must only store a single pointer per spanning tree (the gradient direction). An execution packet's ability to find the appropriate resources depends on several fabrication variables, including defect rates and the distribution of node types (evaluating this space is future work). In the next subsection, we describe how we can exploit the packet-routing infrastructure to configure a fully addressable memory system in each cell.

4.5 Memory

Each cell represents a local namespace for memory and includes both data and instructions. The memory system must be able to (a) allocate (number or name) its locations, (b) provide an interface to execution packets, and (c) route memory packets (both requests to specified locations and responses back to requestors).

4.5.1 *Memory Allocation.* The memory network is a spanning tree rooted at the cell anchor. To configure memory, allocation packets are injected from the via through the anchor node, initially routed on virtual channel zero using any planar gradient. When an unallocated memory node receives an allocation packet, it records the address, marks itself as allocated, and sinks the packet. The second allocation packet received by this node is forwarded along the specified gradient, forming a branch in the network. For the third allocation packet, the node modifies the header to route the packet on virtual channel one along a planar gradient that creates a second branch in the network. Three-fourths of the subsequent allocation packets arriving on virtual channel zero are forwarded along the first branch, while the remaining packets use the other branch and switch to virtual channel one. Packets on virtual channel one are never modified. Cycles in the memory network are prevented by having an allocated node only accept configuration packets on the same physical link as its original allocation packets.

Memory ports are allocated after memory nodes and must have three good links (excluding the link used by the incoming packet) with three distinct planar gradients. Ports never change an allocation packet gradient, thus keeping the remaining two links free for the execution network. Memory ports are unnamed, except for one port where we initiate execution. Non-memory nodes between memory nodes route allocation packets according to the specified gradient and reserve the corresponding links only for memory operations. A second planar gradient configuration creates new spanning trees that do not include any of the memory network links, thus creating two separate networks. Figure 8 illustrates the allocation of 64 memory locations and 64 ports on a 32×32 grid with a 3% defect rate. For illustration only, we include only the West planar gradient on the execution network and use a low defect rate on a small grid. Clearly, in this memory system the anchor node could be a bottleneck.

4.5.2 *Interfacing Execution and Memory.* The interface between the execution network and the memory network is controlled by memory ports that assume responsibility for handling all memory operations, including the JMP/CALL instructions for packet instantiation (see Section 4.6). When an execution packet needs to perform a constant or indirect memory operation, it searches for a memory port. A memory port servicing an execution packet stalls the execution packet, but at different points for loads and stores. Since load addresses are contained in the instruction field, the load can immediately issue and only stall the packet when the first bit of the operand stream arrives. Thus, the header continues searching for resources for subsequent instructions. When the memory port that initiated the load receives the response, it interleaves the memory contents into the execution packet's operand stream, enabling the operand stream to continue forward. A store must see the entire operand stream to extract the data, and after the node issues the store, it stalls the packet until the store is acknowledged. This acknowledgment ensures inter-packet memory disambiguation. Memory ports also support indirect memory operations which require back-to-back memory operations: one to load the address, and the other to access the contents at that address. We implement this

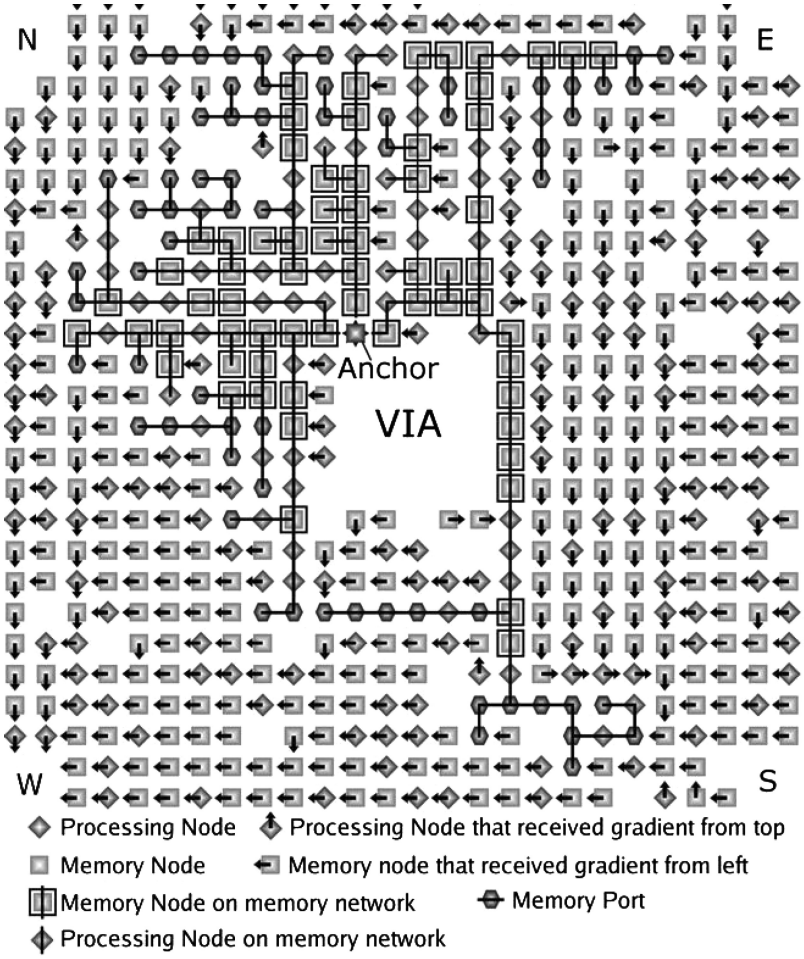


Fig. 8. Memory network. 32×32 grid with a fully configured memory network, showing one gradient (West).

by first issuing a constant load to obtain the address, and then using the result to generate another address for the load or store.

4.5.3 *Routing Memory Packets.* Memory packets are routed on either a request or response virtual network (two virtual channels per unidirectional link) that each obey up*/down* routing. Routing in the up direction follows the cell gradient up the spanning tree to the anchor node where the packet is broadcast in the down direction. Broadcasting is necessary since the destination memory node or port could be anywhere in the memory network. Loads require two full traversals of the memory network. However, since the anchor node is a serialization point for memory operations, it can acknowledge a store by broadcasting down the response network. Memory operations for addresses outside the originating cell are passed by the anchor onto the microscale network.

4.6 Packet Instantiation and Chaining

Entire execution packets (from header through tail) can be stored in memory by fragmenting them across memory nodes. Each fragment contains a portion of the execution packet and the memory address of the next sequential fragment (zero indicates termination). The fragments are written into memory using the micro-scale interface to inject store requests into the memory network. Packets are reassembled and instantiated on the execution network at a memory port using special sequencing instructions. Initial execution starts by using the micro-scale interface to inject one of these instructions on the memory response network for the named memory port.

Chaining is the process of sequencing instructions or packets under software control by including a special instruction as the last operation. We implement two forms of the sequencing instruction: (1) CALL creates an entirely new packet, but stalls until all previous instructions are complete (i.e., it sees the packet tail), and (2) JMP injects new operations into the existing packet by stalling the operand stream, thus enabling accumulator forwarding. Conditional CALL is easily supported since the instruction waits for the packet tail. Execution of one packet can overlap with its dependent packet's search for functional and memory nodes. We leave full exploration of the instruction set and various forms of parallelism as future work.

4.7 Improving Node Utilization

While the four planar gradients allow us to route execution packets in the cell, we find that only a small fraction of all execution resources in a cell are used. This is because the route taken by the execution packet depends on its insertion point in the cell and the gradient that is being used to route. The execution network within the cell does not have a well-defined structure if we use planar gradients for routing. To improve the number of nodes reachable by execution packets, we need to modify the structure of the execution network within a cell.

We add a second via and anchor node ("execution anchor") to the cell. This via is used only by the execution network. Once the memory system has been created, we broadcast an "execution" gradient in the cell. This gradient reaches nodes that have not been included in the memory network and any ports on the memory network. This allows us to create an execution network with better structure by performing a depth-first traversal on the spanning tree created during the broadcast of the execution gradient. All execution packets follow this depth-first order, ensuring high execution node utilization. The memory and execution networks now include most of the nodes in the cell, potentially allowing the use of about 97% of the cell (some nodes can become isolated during the creation of the memory network). However, as we discuss in Section 5.5, there are other aspects of NANA that limit node utilization.

5. PRELIMINARY EVALUATION

This section presents a preliminary evaluation of NANA. Our goal is to demonstrate the viability of the approach and to provide more details on execution.

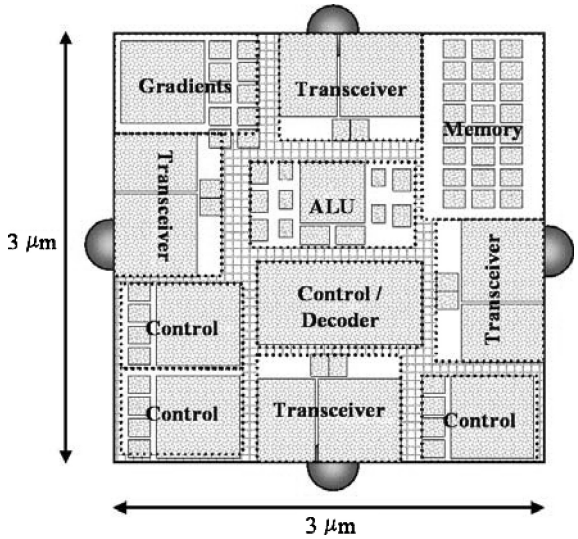


Fig. 9. Node floorplan.

CNFET device characteristics suggest that this technology may have significant advantages over silicon in terms of power, delay, and cost. We are collaborating with physical scientists to fabricate and characterize CNFET electronics, which will enable quantitative evaluation. We have developed a tool chain to support automated circuit design [Dwyer et al. 2004b, 2004c] and architectural evaluation. We first present an initial node floorplan, describe our simulation framework, and then demonstrate system operation and provide preliminary performance results using two simple programs: (1) Fibonacci is strictly an illustrative example, and (2) string matching reveals the potential to exploit massively parallel computation with nanoelectronics. We conclude the section with an analysis of the strengths and weaknesses of the proposed design.

5.1 Node Floorplan

Figure 9 shows an initial floorplan for a 3μm $\times$ 3 μm node that includes both an ALU and 16 bits of data storage with 8-bit addresses. The four semicircles around the node represent contact points for inter-node links. The four transceivers control data transfer between the node and its neighbors. Configuration and gradient state is stored in the block denoted 'Gradients,' while control logic is distributed in the four blocks marked 'Control,' one of which is also responsible for decoding instructions (marked 'Control/Decoder'). The small, unlabeled blocks next to transceivers are the interfaces between the transceiver and the control/data logic of the node. The largest area is consumed by the various state machines sized according to the requirements derived from our simulator. Our current implementation assumes specialized nodes, enabling more area for control and buffering.

Table IV. Packet Layout

Address	Op	Next		Address	Op	Next
0×10	LD (0×30)	0×14		0×26	CPACC	0×28
0×14	LD (0×32)	0×18		0×28	ADD	$0 \times 2A$
0×18	LD (0×34)	$0 \times 1A$		$0 \times 2A$	CST (0×36)	$0 \times 2E$
$0 \times 1A$	DEC	$0 \times 1C$		$0 \times 2E$	ST (0×34)	0×32
$0 \times 1C$	CMPZ	0×20		0×32	ST (0×32)	0×36
0×20	ST (0×30)	0×24		0×36	CALLZ (0×10)	0×0
0×24	SWAP	0×26				

5.2 Simulation Framework

We evaluate NANA using a custom event-driven simulator written in C++ that simulates the system in detail. The simulator models activity within each node down to bit exchanges between components. The simulator models all node types and the system at all stages, including gradient broadcast, memory configuration, execution configuration and run-time. It allows the user to vary a number of system parameters, including the size of the network, node type distribution, event latencies, defect rate, and number of cells being simulated. Each cell holds a different part of the global address space and can execute different programs that are provided as input to the simulator. All events in the simulator are assumed to be a multiple of the clock-cycle time (0.1 ns). The simulator accepts user-defined network topologies, or it can generate regular grid-based topologies. For simplicity, we use a grid-based topology with a single 1024 node cell and a 3% node defect rate in our evaluation. As long as the defect rate is low (about 15% or lower), the network topology has little effect on performance.

5.3 Fibonacci

In this section we consider the simple code that computes the Nth Fibonacci number. Table IV shows the packet needed to implement Fibonacci for N $>= 1$ (N is stored at address 0×30), and how the fragments are arranged in memory. For simplicity, each instruction is a separate fragment. The first packet, starting at address 0×10, loads the value N (counter) which specifies which Fibonacci number to compute, and the constants 1 and 0 (pre-loaded into 0×32 and 0×34 to begin with). The fourth instruction decrements the counter and sets the condition bit in the tail if the counter is zero. The counter is then stored back at address 0×30. The seventh instruction swaps the first two operands in the operand stream. The eighth instruction creates a copy of the accumulator at the end of the operand stream. The ninth instruction (ADD) computes the next Fibonacci number. If the condition flag in the tail is set, this new computed value is stored at address 0×36. The two remaining operands are then stored at locations 0×34 and 0×32. Finally, if the condition flag is not set, we loop back to the beginning using a CALLZ instruction, creating a new packet. If the condition flag is set, the instruction is not executed, terminating the program. Figure 10 illustrates the creation of this packet with a bootstrapping JMP. In Figure 10(a), we show the bootstrapping packet inserted at the via in the execution network. This packet is routed along the execution network until it

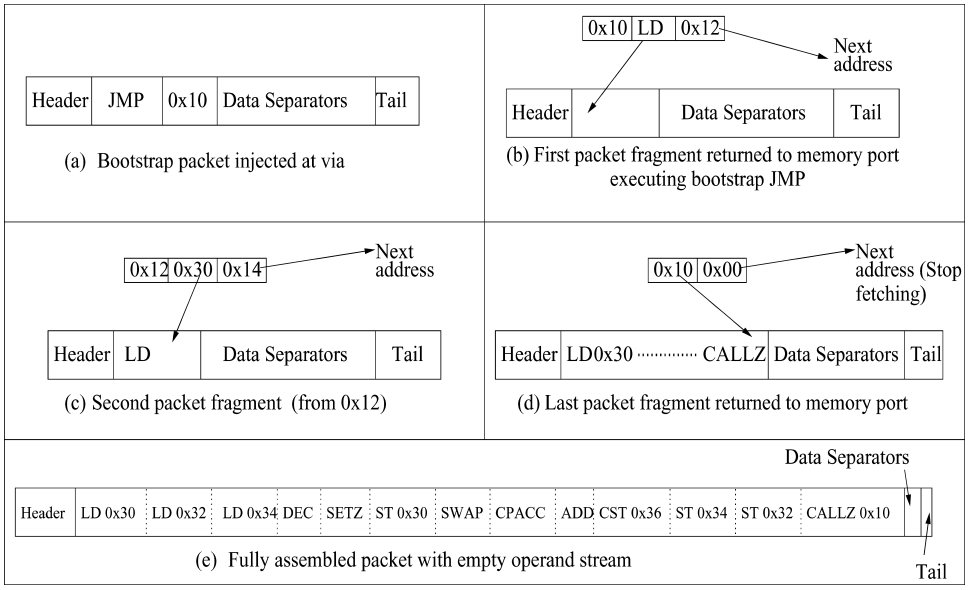


Fig. 10. Bootstrapping the fibonacci execution packet with a JMP.

finds a memory port. The JMP instruction in the packet executes at the port and starts fetching data from location 0×10 (where the Fibonacci code is stored). The data returned from location 0×10 (Figure 10(b)) is divided into two parts: (1) data for the packet and (2) next address. The data for the packet (in this case, a LD opcode) is inserted into the packet and sent out on the execution network. The next address is used to fetch the next fragment of code (in this case, from address 0×12). The data returned from location 0×12 (Figure 10(c)) provides the address for the LD instruction and the address of the next fragment of code. This process is repeated until we get a data fragment back with 0×00 as the next address (Figure 10(d)). This indicates that we have finished executing the JMP instruction. The final packet before execution begins is shown in Figure 10(e). It is important to note that execution can begin while the JMP instruction is still executing.

To demonstrate our system operation, we simulate its behavior at the bit-serial link level executing the above packets. We model a single 32×32 cell with 25% ALU nodes and four corner vias for planar gradients. We assume a random distribution of defective nodes, with 3% of all nodes being defective. The memory system in the cell includes 64 16-bit memory nodes and 80 ports. A system using a depth-first execution network would achieve similar performance (depth-first execution only increases the number of nodes reachable on the execution network). The average time per loop iteration (0×10 to 0×36) is 22,300 cycles and it might be possible to reduce this through loop unrolling. However, only 2,000 of the 22,300 cycles are spent in performing the actual computation. More than 20,000 cycles are spent in accessing the memory system. Figure 11 illustrates the execution of the program. We take a snapshot of execution before the first load operation completes. While the absolute performance of this

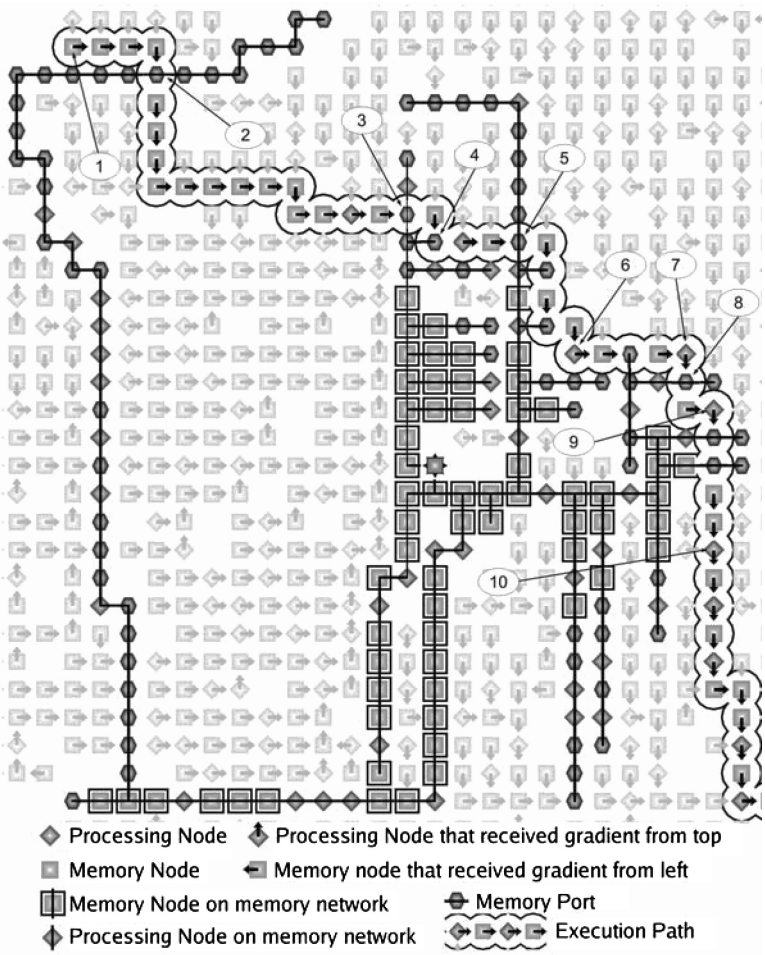


Fig. 11. The path of Fibonacci code in one direction through configured network with 1024 nodes and 3% defects. Unused nodes in the execution network appear faded, defective nodes are omitted. 1: Bootstrap packet injected via; 2: JMP executes at port; 3: LD 0×30 executes at port; 4: LD 0×32 executes at port; 5: LD 0×34 executes at port; 6: DEC at processing node; 7: CMPZ executes at processing nodes; 8: ST 0×30 executes at port; 9: SWAP executes at processing node; 10: ADD executes at processing node.

example does not surpass even current CMOS, it serves to demonstrate the operation of a single cell. The greatest advantage of this technology arises from the scale of the system.

5.4 String Match

The opportunity for massively parallel computation is tremendous. String searching is a common operation in many applications (e.g., searching for particular DNA sequences within a genome). Our string-match program loads a 16-bit key and compares it to all data elements within the cell, and a conditional store indicates if a match was found. This implementation requires 48 memory

locations for instructions and 16 for data. Therefore, we can search a 32GB database by using all 10^9 cells. The execution time within one cell is 35 ns per comparison, for a total of 28.5×10^6 comparisons/sec. The potential for massive parallelism would be exposed by having each of the 10^9 cells perform a unique comparison, yielding an overall rate of 2.85×10^{16} comparisons/sec.

5.5 Discussion

The peak performance of NANA (assuming 1/2 the nodes compute) is significantly higher than today's supercomputers. NANA can potentially perform 4.12×10^{21} bitops/sec while the IBM Blue Gene can achieve 4.6×10^{16} bitops/sec and the NEC Earth Simulator can achieve a peak of 5.2×10^{15} bitops/sec. However, it will be a challenge for NANA to realize this peak performance in practice. Developing these programs exposes two key limitations of our current architecture: (1) under-utilization of nodes and network connectivity, and (2) bottlenecked memory system.

Under-utilization of nodes. One of the key limiting factors to achieving good performance is the fact that nodes spend only a small fraction of their time doing useful work. For example, if we are executing 10 arithmetic instructions, the node that executes the first instruction is doing useful work only when (a) it is receiving the first instruction and (b) it is receiving its operands for execution. Since there are 10 instructions being executed which will require 11 operands (assuming data is pre-loaded), the packet will contain 868 bits (including header, instructions, operands, field separators and tail). Out of these, only 220 bits (header, instruction to be executed, separators, two operands, the operand separators and tail) are relevant to the execution of the instruction. Thus, the node is doing "useful" work only when it is dealing with ∼25% of the bits in the packet. No useful computation is performed by the node in the remaining time.

The depth-first execution network increases the number of nodes usable during execution, but it does not reduce node idle time. The execution network can be thought of as a pipeline of nodes. The pipeline is most efficient only when it is full. Similarly, the execution network is fully utilized only when all nodes are actively executing instructions. This would require the creation of extremely long packets. However, the longer the packet, the longer it takes for a node to execute instructions because longer packets typically have longer instruction and data fields and a node needs to forward the entire packet before it can handle the next packet. This limits the peak performance of NANA.

Memory system bottleneck. The memory system in NANA has multiple bottlenecks. Because of the way it is designed, it is currently not possible to execute store instructions (direct, indirect, or conditional) from a packet before any load instructions (direct, or indirect). This limits the size and content of execution packets that can be created. In addition, all memory requests are serialized through the anchor node. This creates a substantial bottleneck at the anchor node. There is no easy way of alleviating this bottleneck without significantly adding to the complexity of the system. Finally, our limited routing capability in the random network limits our ability to build a

balanced memory network. This often results in unbalanced networks with long latencies.

Despite its limitations, NANA demonstrates that it is possible to build a computing system despite the severe technological constraints. As a first step, NANA does remarkably well. Future designs based on this technology can use the insights gained from this design. We believe that NANA is a necessary first step toward exploiting nanotechnology's potential to overcome the "red brick wall."

6. RELATED WORK

This article covers a wide range of topics from novel bottom-up fabrication techniques and emerging devices through microarchitecture and instruction set design. There is significant literature on most of these important topics. However, to keep focus we discuss only the most closely related architecture work.

The most related work is Dwyer's proposal to use a DNA guided self-assembly technique to build a massively parallel computer [Dwyer 2003; Dwyer et al. 2004a, 2004b]. The proposed machine has no communication between processing elements, thus targets problems that are "embarrassingly parallel." By contrast, our work aims to build a more conventional processor. Goldstein's work on nanofabrics leverages reconfigurable self-assembled nanoelectronics to provide a defect tolerant architecture [Goldstein and Budiu 2001]. Resonant tunneling diodes (two terminal devices) are configured into supernodes of appropriate functionality after a test phase maps out defective components. The nanofabric is reconfigured for each program that executes. DeHon presents an architecture that exploits three terminal devices (FETs) by self-assembling arrays of nanowires and FETs [DeHon 2003]. Sparing and remapping are used to provide defect tolerance. Other research investigates defect tolerant architectures [Han and Jonker 2003; Han et al. 2005; Heath et al. 1998; Nikolik et al. 2002; Snider et al. 2004; Thaker et al. 2005], various array-based nanoarchitectures [Ancona 1996; Beckett and Jennings 2002; Fountain et al. 1998] as well as alternative emerging nanoelectronic technologies [Gayasen et al. 2005; Niemier and Kogge 2001; Oskin et al. 2002; Tour 2000; Tseng and Ellenbogen 2001], and some of the challenges that will be faced in dealing with these emerging technologies and possible means of overcoming those challenges [Fortes 2003; Niemier et al. 2004; Stan et al. 2003].

7. CONCLUSIONS

This article presents an architecture that addresses the challenges posed by DNA-based self-assembly of carbon nanotubes and other nanotechnologies with similar characteristics (possibly even scaled CMOS). To overcome (1) limited node size, (2) random interconnection of nodes, and (3) a high defect rate, we developed an active-network architecture with an accumulator-based ISA. This architecture enables execution packets to search through a sea of heterogeneous nodes for the functionality they need, while avoiding defective nodes. We use an initial configuration phase to impose some limited structure on the computing

substrate, particularly for routing and memory allocation. We simulate this architecture running simple programs to demonstrate its viability, and provide preliminary performance numbers. While this architecture is only a relatively unoptimized first step, it addresses some of the key challenges in this class of nanotechnology and it highlights the technology's architectural implications. There is a significant amount of future work, including fabrication, layout, ISA design, supporting speculation and parallelism, etc.

REFERENCES

ANCONA, M. G. 1996. Systolic processor designs using single-electron digital circuits. *Superlattices and Microstructure 20*, 4.

BACHTOLD, A., HADLEY, P., NAKANISHI, T., AND DEKKER, C. 2001. Logic circuits with carbon nanotube transistors. *Science 294* (Nov.), 1317–1320.

BECKETT, P. AND JENNINGS, A. 2002. Toward nanocomputer architecture. In *Proceedings of the 7th Asia-Pacific Computer Systems Architecture Conference.* 141–150.

BRAUN, E., EICHEN, Y., SIVAN, U., AND GDALYAHU, B.-Y. 1998. DNA-templated assembly and electrode attachment of a conducting silver wire. *Nature 391*, 775–778.

BURKE, P. J. 2003. An RF circuit model for carbon nanotubes. *IEEE Trans. Nanotech 2*, 1, 55–58.

CASTRO, M. AND LISKOV, B. 1999. Practical byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*.

CAMPBELL-KELLY, M. 1998. Programming the edsac: Early programming activity at the University of Cambridge. *IEEE Annals History Compt. 20*, 4, 46–67.

CUI, Y. AND LIEBER, C. M. 2001. Functional nanoscale electronic devices assembled using silicon nanowire building blocks. *Science 291* (Feb.), 851–853.

CULBERTSON, W. B., AMERSON, R., CARTER, R. J., KUEKES, P., AND SNIDER, G. 1996. 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*. (Nov.).

DALAL, Y. K. AND METCALFE, R. M. 1978. Reverse path forwarding of broadcast packets. *Commun. ACM. 21*, 12 (Dec.), 1040–1048.

DALLY, W. J. 1992. Virtual channel flow control. *IEEE Trans. Parallel and Distributed Systems 3*, 2 (Mar.), 194–205.

DEHON, A. 2002. Array-based architecture for molecular electronics. In *Proceedings of the 1st Workshop on Non-Silicon Computation (NSC-1)* (Feb.).

DEHON, A. 2003. Array-based architecture for FET-based, nanoscale eletronics. *IEEE Trans. Nanotech. 2*, 1 (Mar.), 23–32.

DÜRKOP, V., GETTY, S. A., COBAS, E., AND FUHRER, M. S. 2004. Extraordinary mobility in semiconducting carbon nanotubes. *Nano Letters 4*, 1, 35–39.

DWYER, C., GUTHOLD, M., FALVO, M., WASHBURN, S., SUPERFINE, R., AND ERIE, D. 2002. DNA functionalized single-walled carbon nanotubes. *Nanotech. 13*, 601–604.

DWYER, C. 2003. Self-Assembled Computer Architecture: Design and Fabrication Theory. PhD thesis, University of North Carolina, May.

DWYER, C., VICCI, L., POULTON, J., ERIE, D., SUPERFINE, R., WASHBURN, S., AND TAYLOR, R. M. 2004. The design of DNA self-assembled computing circuitry. *IEEE Trans. VLSI 12* (Nov.), 1214–1220.

DWYER C., CHEUNG, M., AND SORIN, D. J. 2004. Semi-Empirical spice models for carbon nanotube FET logic. In *Proceedings of the 4th IEEE Conference on Nanotechnology* (Aug.).

DWYER, C., JOHRI, V., PATWARDHAN, J. P., LEBECK, A. R., AND SORIN, D. J. 2004. Design tools for self-assembling nanoscale technology. *Institute of Physics Nanotech. 15*, 9 (Sept.).

DWYER, C., POULTON, J., TAYLOR, R., AND VICCI, L. 2004d. DNA self-assembled parallel computer architectures. *Nanotech.* 1688–1694.

DWYER, C., PARK, S. H., LABEAN, T., AND LEBECK, A. 2005. The design and fabrication of a fully addressable 8-tile DNA lattice. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*. 187–191.

FORTES, J. B. 2003. Future challenges in vlsi system design. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI* (Feb.), 5–7.

FOUNTAIN, T. J., DUFF, M. J. B., CRAWLEY, D. G., TOMLINSON, C. D., AND MOFFAT, C. D. 1998. The use of nanoelectronic devices in highly-parallel computing systems. *IEEE Transactions on VLSI Systems 6*, 1, 31–38.

FUHRER, M. S., NYGARD, J., SHIH, L., FORERO, M., YOON, Y.-G., MAZZONI, M. S. C., CHOI, H. J., IHM, J., LOUIE, S. G., ZETTLE, A., AND MCEUEN, P. L. 2001. Crossed nanotube junctions. *Science 288* (Apr.), 494–497.

GAYASEN, A., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2005. Exploring technology alternatives for nano-scale FPGA interconnects. In *Proceedings of the 42nd Annual Design Automation Conference (DAC-2005)*, (June).

GLASS, C. J. AND NI, L. M. 1992. The turn model for adaptive routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May), 278–287.

GOLDSTEIN, S. C. AND BUDIU, M. 2001. Nanofabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (July), 178–191.

HAN, J. AND JONKER, P. 2003. A defect- and fault-tolerant architecture for nanocomputers. *Nanotech. 14* (Jan.), 224–230.

HAN, J., GAO, J., QI, Y., JONKER, P., AND FORTES, J. A. B. 2005. Toward hardware-redundant, fault-tolerant logic for nanoelectronics. *IEEE Design & Test of Computers 22*, 4 (Apr.), 328–339.

HAZANI, M., HENNRICH, F., KAPPES, M., NAAMAN, R. N., PELED, D., SIDOROV, V., AND SHVARTS, D. 2004. DNA-mediated self-assembly of carbon nanotube-based electronic devices. *Chemical Physics Letters 391*, 389–392.

HEATH, J. R., KUEKES, P. J., SNIDER, G. S., AND WILLIAMS, R. S. 1998. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*. *280* (June), 1716–1721.

HUANG, Y., DUAN, X., CUI, Y., LAUHON, L. J., KIM, K.-H., AND LIEBER, C. M. 2001. Logic gates and computation from assembled nanowire building blocks. *Science 294* (Nov.), 1313–1317.

INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. *Mobile Comput. Networking*, 56–67.

JAVEY, A., GUO, J., FARMER, D. B., WANG, Q., AND WANG, D., 2004. Carbon nanotube field-effect transistors with integrated ohmic contacts and high-K gate dielectrics. *Nano Letters 3*, 447–450.

JOHNSON, D. B. AND MALTZ, D. A. 1996. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, vol. 353. (Imielinski and Korth, eds.). Kluwer, Amstredam.

KIM, H.-S. AND SMITH, J. E. 2002. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (May).

KIM, H.-S. AND SMITH, J. E. 2003. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO) 2003* (Mar.), 25–35.

KIM, B. M., BRINTLINGER, T., COBAS, E., FUHRER, M. S., ZHENG, H., YU Z., DROOPAD, R., RAMDANI, J., AND EISENBEISER, K. 2004. High-performance carbon nanotube transistors on $SrTiO_3$ Si substrates. *Applied Physics Letters 84*, 11, (Mar.).

LAVINGTON, S. H. 1978. The manchester Mark 1 and Atlas: A historical perspective. *Communications of the ACM 21*, 1, 4–12.

LIU, D., PARK, S.-H., REIF, J. H., AND LABEAN, T. H. 2004. Dna nanotubes self-assembled from TX tiles as templates for conductive nanowires. In *Proceedings of the National Academy of Science 101*, 3, 717–722.

MARTIN, B. R., DERMODY, D. J., REISS, B. D., FANG, M., LYON, L. A., NATAN, M. J., AND MALLOUK, T. E. 1999. Orthogonal self-assembly on colloidal gold-platinum nanorods. *Advanced Materials 11*, 12 (Aug.), 1021–1025.

MCEUEN, P. L., FUHRER, M. S., AND PARK, H. 2002. Single-walled carbon nanotube electronics. *IEEE Trans. Nanotech. 1*, 1 (Mar.), 78–85.

NIEMIER, M. T. AND KOGGE, P. M. 2001. Exploring and exploiting wire-level pipelining in emerging technologies. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (July), 166–177.

NIEMIER, M. T., RAVICHANDRAN, R., AND KOGGE, P. M. 2004. Using circuits and systems-level research to drive nanotechnology. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)* (Oct.), 302–309.

NIKOLIC, K., SADEK, A., AND FORSHAW, M. 2002. Fault-tolerant techniques for nanocomputers. *Nanotech. 13*, 357–362.

OSKIN, M., CHONG, F. T., AND CHUANG, I. 2002. A practical architecture for reliable quantum computers. *IEEE Computer* (Jan.), 79–87.

PARK, S. H., PISTOL, C., AHN, S. J., REIF, J. H., LEBECK, A. R., DWYER, C. L., AND LABEAN, T. H. 2006. Finite-size, fully-addressable DNA tile lattices formed by hierarchical assembly procedures. *Angewandte Chemie 45* (Jan.), 735–739.

PATWARDHAN, J. P., DWYER, C., LEBECK, A. R., AND SORIN, D. J. 2004. Circuit and system architecture for DNA-guided self-assembly of nanoelectronics. In *Foundations of Nanoscience: Self-Assembled Architectures and Devices*. 344–358.

PATWARDHAN, J. P., DWYER, C., LEBECK, A. R., AND SORIN, D. J. 2005. Evaluating the connectivity of self-assembled networks of nano-scale processing elements. In Proceeding of the *IEEE International Workshop on Design and Test of Defect-Tolerant Nanoscale Architectures (NANOARCH '05)*, (May), 2.1–2.8.

SCHROEDER, M. D., BIRRELL, A. D., BURROWS, M., MURRAY, H., NEEDHAM, R. M., RODEHEFFER, T. L., SATTERTHWAITE, E. H., AND THACKER, C. P. 1991. Autonet: A high-speed, self-configuring local area network using point to point links. *IEEE Journal on Selected Areas in Communications 9*, 8, (Oct.).

SEEMAN, N. C. 1999. Dna engineering and its application to nanotechnology. *Trends in Biotech 17*, 437–443.

SKINNER, K., CARROLL, R. L., WASHBURN, S., AND DWYER, C. L. 2005. Nanowire transistors, gate electrodes, and their directed self-assembly. In *Proceedings of the 72nd Southeastern Section of the American Physical Society (SESAPS)* (Nov.).

SNIDER, G., KUEKES, P., AND WILLIAMS, R. S. 2004. CMOS-like logic in defective, nanoscale crossbars. *Nanotech. 15*, 881–891.

STAN, M. R., FRANZON, P. D., GOLDSTEIN, S. C., LACH, J. C., AND ZIEGLER, M. M. 2003. Molecular electronics: From devices and interconnect to circuits and architecture. In *Proc. IEEE. 91*, (Nov.), 1940–1957.

STRANO, M. S., DYKE, C. A., USREY, M. L., BARONE, P. W., ALLEN, M. J., SHAN, H. W., KITTRELL, C., HAUGE, R. H., TOUR, J. M., AND SMALLEY, R. E. 2003. Electronic structure control of single-walled carbon nanotube functionalization. *Science 301* (Sept.), 1519–1522.

TANS, S. J., VERSCHUEREN, A. R. M., AND DEKKER, C., 1998. Room-temperature transistor based on a single carbon nanotube. *Nature 393*, 49–52.

TENNENHOUSE, D. L. AND WETHERALL, D. J. 1996. Towards an active network architecture. *Computer Communication Review 26*, 2.

THAKER, D. D., IMPENS, F., CHUANG, I. L., AMIRTHARAJAH, R., AND CHONG, F. T. 2005. Recursive tmr: Scaling fault tolerance in the nanoscale era. *IEEE Design & Test of Comput. 22*, 4 (Apr.), 298–305.

TOUR, J. M. 2000. Molecular electronics. Synthesis and testing of components. *Accounts of Chemical Research 33*, 11, 791–804.

TSENG, G. Y. AND ELLENBOGEN, J. C. 2001. Toward nanocomputers. *Science 294* (Nov.), 1293–1294.

BERKEL, K. V. AND BINK, A. 1996. Single-track handshake signaling with application to micropipelines and handshake circuits. In *Procceding of the Seconds International Symposium on Advanced Research in Asynchronous Circuits and Systems*. (Mar.), 122–133.

WIND, S. J., APPENZELLER, J., MARTEL, R., DERYCKE, V., AND AVOURIS, P. 2002. Vertical scaling of carbon nanotube field-effect transistors using top gate electrodes. *Applied Physics Letters 80* (May), 3817–3819.

WINFREE, E., LIU, F., WENZLER, L. A., AND SEEMAN, N. C. 1998. Design and self-assembly of two-dimensional DNA crystals. *Nature* 394, 539.

YAN, H., LABEAN, T. H., FENG, L., AND REIF, J. H. 2003a. Directed nucleation assembly of barcode patterned DNA lattices. *Proceedings of the National Academy of Sciences 100*, 14 (July), 8103–8108.

YAN, H., PARK, S. H., FINKELSTEIN, G., REIF, J. H., AND LABEAN, T. H. 2003b. DNA templated self-assembly of protein arrays and highly conductive nanowires. *Science 301*, 5641 (Sept.), 1882–1884.

ZHENG, M., JAGOTA, A., SEMKE, E. D., DINER, B. A., MCLEAN, R. S., LUSTIG, S. R., RICHARDSON, R. E., AND TASSI, N. G. 2003. DNA-assisted dispersion and separation of carbon nanotubes. *Nature Materials 2* (May), 338–342.

Received August 2005; revised February 2006; accepted February 2006