Programmable Nanocontrollers For Nanodevices

H. G. Dietz
Electrical & Computer Engineering Department
University of Kentucky
Lexington, KY 40506-0046
hankd@engr.uky.edu

Advances in nanotechnology have made it possible to assemble nanostructures into a wide range of micrometer-scale sensors, actuators, and other novel devices... and to place thousands of such devices on a single chip. Most of these devices can benefit from intelligent control, but the control often requires full programmability for each device's controller. Routing thousands of signals off-chip is not practical, but neither are conventional microcontroller designs able to be made small enough to be paired with each of the thousands of devices on-chip. This paper outlines an approach toward achieving the goal of building fully programmable controllers with circuit complexity low enough to allow each nanotechnology-based device to be accompanied by its own **nanocontroller**.

1. Motivation

The considerable body of work in nanoprocessors centers on using new nanotechnology implementation technologies to produce either very complex conventional processors or novel types of massively parallel computing devices. Our goal is not computing per se, but intelligent control of devices built using nanotechnology. Our concept of a **nanocontroller** is a programmable controller that is simple enough to be on the same chip as, and paired with, the micrometer-scale nanotechnology-based device(s) that it controls. These nanocontrollers also would be appropriate as embedded controllers for Micro-Mechanical Devices (MMD) and other somewhat larger devices.

It is hard to imagine a world without intelligent control of human-scale devices, but this is a very recent phenomenon. The potential impact of nanocontrollers on nano- and microscale devices is equally strong.

As a hypothetical example, consider chemical and/or biological sensor array chips. In military or homeland security applications, a common goal of nanotechnology efforts is to create a single chip sensor array that can detect and measure the levels of a wide range of chemical and biological toxins. The usual vision of the system is something like:

This pipeline appears to be necessary to accommodate significant digital post-processing requirements. Some of the processing might be correcting for nonlinear errors in the analog sensor outputs; these errors might vary from sensor chip to sensor chip, thus requiring individual units to be calibrated. Subsequent processing might analyze the corrected sensor levels to determine the level of health threat, report the levels of relevant sensors, and select the appropriate remedial action to be taken, e.g., choice of various antidotes and protective gear.

Instead, suppose that each sensor is accompanied by a nanocontroller that could be programmed to apply the calibrated nonlinear correction. Using some low-temperature nanofabrication technologies to build the sensor array, it might be possible to first create a chip full of nanocontrollers and then create the sensor array as a layer above the nanocontrollers — literally taking no additional space on the chip to add this intelligence. The nanocontrollers under the sensors, perhaps with additional nanocontrollers on the same chip, can then operate as a parallel computer system to evaluate threats. The results would be directly output from the sensor chip — perhaps as digital signals directly displaying the appropriate remedial action on a Liquid Crystal Display (LCD) or as playback of the appropriate digitized audio announcement.

In summary, the nanocontroller solution would be smaller, more durable, cheaper, consume less power, and very likely provide higher performance.

In addition to application within a wide variety of smart chips using nanofabrication technology, notice that we define nanocontrollers by their circuit complexity, not by their physical size. Physically large nanocontrollers made using organic semiconductors or similar technologies might be useful in applications such as smart control of pixels in a large-scale display.

2. What Must A Nanocontroller Be Able To Do?

The vision of nanocontrollers embedded alongside nanotechnology devices cannot be implemented using conventional microcontroller architectures and compilation technology. It will be enabled by developing a new set of architectural and compilation technologies that together can satisfy the basic requirements for such a system. There are six primary requirements, outlined in the following subsections.

2.1. Minimal Circuit Size

The circuit complexity per nanocontroller must be low enough to be comparable in physical size to sensors, actuators, and other devices implemented using nanotechnology. Even the simplest microcontrollers generally require *thousands of gates*; our goal is to reduce that number to no more than *hundreds of transistors*.

2.2. Predictable Real-Time Behavior

From a programmer's point of view, a nanocontroller must have predictable real-time execution timing characteristics. In order to monitor or control the real-time behavior of a device, it often will be necessary for the nanocontroller to perform particular operations at precise times relative to other operations. Although some nanotechnology devices can tolerate very slow controller time bases, the required timing precision varies greatly depending on the type of device with which the nanocontroller must interact. The small physical scale of some devices results in relatively small time constants. As an initial goal, a nanocontroller should be able to handle real-time constraints with accuracies no worse than a microsecond. Fortunately, the simplicity of nanocontrollers should allow instruction execution times of a nanosecond or less, so microsecond timing accuracy should be easy to achieve.

2.3. Localized Input/Output

Each nanocontroller must be be able to perform appropriate digital and/or analog input/output (I/O) operations to interact with the nanotechnology device(s) with which it is associated.

Digital I/O may be as simple as having some memory cells or registers be input/output devices.

Analog I/O is substantially more complex. Many nanotechnology devices have inherently analog interfaces, and the space required for separate Analog-to-Digital Converter (ADC) or Digital-to-Analog Converter (DAC) units would be too great. Thus, an analog input would most likely be implemented by timing (in software) how long it took for a digital threshold voltage to be crossed in charging a capacitor. An analog output can be accomplished by a similar process, essentially using Pulse-Width Modulation (PWM) software to drive a simple filter circuit. These approaches also help in that using a separate ADC/DAC unit tends to fix the precision of conversion, whereas the method discussed permits precision to be traded for sample speed under program control. Of course, this type of analog I/O is possible only with a fast enough processor that also has the predictable timing described in 2.2.

2.4. Coordination As A Parallel Computer

E pluribus unum: each nanocontroller is but one of many that may need to act as one. With thousands or millions of devices on a single chip, each with it own nanocontroller, it often is necessary to coordinate the actions of all the devices or to reduce thousands of sensor inputs to their single higher-level meaning. For example, a chip with a variety of types of analog sensors that are together able to detect or deduce levels of thousands of different chemical compounds might only need to report the action that the user should take to counter the set of chemical or biological agents currently sensed. Thus, the nanocontrollers must be able to act together as a parallel computing system.

2.5. Each Nanocontroller Independently Programmable

Each nanocontroller must be fully programmable as an independent processor. Although nanocontrollers may need to work together, control and sensing algorithms often require different constants or even different code paths depending on the state of the device with which each nanocontroller interacts. Basic properties of nanotechnology-based devices make this divergence even more likely, because the variability inherent in the properties of nanoscale devices is generally greater than is commonly seen for larger-scale constructions.

For example, variations in the molecular-level structure of sensors may cause non-linearities to be substantially different for adjacent nanotechnology sensors made with the "identical" design, thus requiring different constants and/or algorithms for normalizing their values. Another reason for independent programmability is to gracefully degrade system performance when some controlled devices are faulty. A faulty device could be given a different control program that acts to minimize the impact of the particular fault present.

Independent programmability implies that there is a programming system that can support coding of arbitrary algorithms. A compiler for a dialect of a familiar programming language, such as C, would be highly desirable.

2.6. Reprogrammability

Nanocontroller programs must be able to be changed easily and perhaps dynamically, but not frequently compared to the speed of their execution. The nanocontrollers may have their program upgraded or changed under various conditions, but self-modifying code is not desirable for this type of control system. Similarly, as a control system rather than a general-purpose computer, it is likely that submission of a new program from outside the system will be infrequent. Thus, it is acceptable and appropriate to perform expensive compile-time transformations to improve the efficiency of each program.

The most likely scenario for reprogramming probably is the detection of development of a sensor fault. For example, it is a well-known phenomenon that image sensors (e.g., CCD arrays) develop bad pixels over a period of years. Even a relatively expensive reprogramming process can be accommodated when new faults arise so infrequently.

3. General Approach

From the basic requirements in section 2, it is possible to begin to focus on a class of valid designs and the architectural and software system technologies they should employ.

3.1. SIMD-Based Architecture

The overriding architectural concern in design of nanocontrollers is 2.1, minimization of the circuit size. A closely related concern has long been a focus in the parallel supercomputing community: the desire to have as many parallel processing elements as possible without exceeding a total system complexity budget. The answer that developed was SIMD (Single Instruction stream, Multiple Data stream).

Currently, driven by the availability of commodity *interchangeable parts* for their construction, MIMD (Multiple Instruction stream, Multiple Data stream) clusters of microprocessor-based nodes are the dominant architecture in parallel supercomputing. Thus, it is easy to forget the long sequence of SIMD supercomputer designs that have been proposed and constructed. Early SIMD supercomputers, especially STARAN [Bat74] and the Goodyear MPP [Bat80], quickly established basic design guidelines. Incremental improvements were made in the NCR GAPP, AMT DAP 510 and 610, Thinking Machines CM1, CM2, and CM200 [TMC89], and MasPar MP1 and MP2 [Bla90], and a multitude of less-well-known SIMD systems. The result is a well-developed body of knowledge about how to architect and use SIMD systems.

SIMD-based architecture is not the only possible answer for nanocontrollers. A wide range of novel architectures have been proposed for nanocomputing, ranging from cellular automata to quantum computing, but it is far less clear how such models can be architected and programmed to support sophisticated control applications. The vast majority of work involving nanotechnology and architecture is not really relevant to producing nano-scale processors, but aims to use nanotechnology to more efficiently implement very complex processors. The primary advantages in using a SIMD architecture are that the concepts are more fully developed and building on existing SIMD work easily meets not just requirement 2.1 above, but also 2.2, 2.4, and 2.6. Adding support for 2.3 is as simple as applying the techniques discussed in section 2.3.

3.2. MIMD-On-SIMD Compiler Technology

What SIMD cannot provide is requirement 2.5, the independent programmability of a MIMD design... or can it? In [Par95], Ken Batcher is quoted as making the point that the circuitry which supports individual programmability of MIMD processors makes a MIMD processing element at least eight times as complex as an otherwise comparable SIMD element. The argument thus was made that an efficiency as low as 12.5% in simulating a MIMD program on SIMD hardware could still favor a SIMD hardware implementation.

It is not surprising that a carefully designed SIMD program can interpretively execute a MIMD program with reasonable efficiency. The interpreter has a data structure, replicated in each SIMD processing element, that corresponds to the internal registers of each MIMD processor. Likewise, each PE's memory holds a copy of the MIMD code to be executed. Hence, the interpreter structure can be as simple as that shown in Figure 1.

The only difficulty in implementing an interpreter is that the simulated machine will be very inefficient. A number of researchers have developed a wide range of “tricks” to produce more efficient MIMD interpreters either through software methods [NiT90], [WiH91], and [DiC92] or through modifying the hardware [Abu97].

1. Each PE fetches an “instruction” into its “instruction register” (IR) and updates its “program counter” (PC).
2. Each PE decodes the “instruction” from its IR.
3. Repeat steps 3a-3c for each “instruction” type:

3.a Disable all PEs where the IR holds an “instruction” of a different type.

3.b Simulate execution of the “instruction” on the enabled PEs.

3.c Enable all PEs.

4. Go to step 1.

Figure 1: Basic MIMD Interpreter Algorithm

Unfortunately, the program for a nanocontroller could be quite complex. Program complexity need not slow the SIMD interpreter’s execution, but it does require that each SIMD processing element have enough local memory to hold its MIMD program code. Many single-chip SIMD computers had few enough processing elements on a chip that a single, reasonably wide, interface to an off-chip RAM could provide perhaps 16KB of local memory per processing element (e.g., as in the MasPar MP1), thus allowing reasonably complex programs to be associated with each processing element. Conversely, placing Processors In Memory (PIM) could provide a modest number of processing elements with significant local memory for each. For nanocontrollers, however, neither of those solutions is viable. To keep nanocontroller size to a minimum, we cannot afford to waste local memory space on holding programs.

Fortunately, an alternative to MIMD-on-SIMD simulation was developed by us in the early 1990s, primarily targeting the MasPar MP1. The technique literally converts the MIMD processor programs into a single SIMD program with similar relative timing properties. We call this process Meta-State Conversion (MSC) [DiK93].

4. Meta-State Conversion (MSC)

In MIMD execution, each processor has its own state. Although these states are generally considered to be independent entities, it also is possible to view the set of processor states at a particular time as single, aggregate, **Meta State**. Using static analysis based on the timing described in [DiZ91], a compiler can convert the MIMD program into an automaton based on meta states, which is directly executed as pure SIMD code.

4.1. Properties Of Meta-State Converted Code

Once a program has been converted into the form of a meta-state automaton, it is no longer necessary for each PE to fetch and decode instructions, nor is it necessary that each PE have a copy of the program in local memory. Only the SIMD control unit needs to have a copy of the meta-state automaton.

Because in execution the meta-state automaton moves from each meta-state to one meta-state successor, there is a single thread of program memory addresses fetched. This makes it relatively easy to utilize a large off-chip RAM to hold the meta-state program. This RAM interface easily can take advantage of nearly all the standard architectural optimizations used to access program memory for modern microprocessors, including caches, branch prediction, and even prefetch. The fact that each meta-state worth of instructions is not just a single instruction, but is an entire block of instructions, also opens the possibility for each block to contain explicit instructions for the SIMD control unit to execute to directly manage caching and prefetch of instruction blocks.

The result is that a SIMD-based nanocontroller can be programmed independently of the other nanocontrollers (2.5). The MSC compiler technology must ensure that timing is preserved (2.2) through this transformation, and changing one processor's program (2.6) is implemented by recompiling the entire set of MIMD programs whenever a new program is introduced.

Just as interpretation has drawbacks, so too does MSC:

- If there are N processors each of which can be in any of S states, then it is possible that there may be as many as $S!/(S-N)!$ states in the meta-state automaton. Without some means to ensure that the state space is kept manageable, the technique is not practical. Fortunately, we have developed a number of techniques that are effective in controlling the state space explosion.
- Meta-state transitions are N-way branches keyed by the aggregate of the MIMD state transitions. This is conceptually simple, but requires some hardware support, e.g., the "global or" of the MasPar MP-1 [Bla90]. There is also a register-allocation-like problem to be solved by the compiler in assigning fixed-size bit masks to distinguish between processors executing different portions of MIMD state within a given SIMD meta state.

4.2. The MSC Algorithm

The first step in MSC of a set of nanocontroller programs is essentially to create a uniform statespace by converting the set of nanocontroller programs into a single MIMD-parallel program. This is trivially accomplished by merging all the independent processor programs into a single MIMD-parallel program which selects the appropriate code for each processor based on a distinguishing parameter. Classically, the distinguishing parameter is a hardwired processing element number, but it can instead be any identifying value that is initialized within each processing element before entering the user program. An example of this transformation is given in Listing 1.

From this point, the algorithm is essentially that which we presented in detail in [DiK93]. However, in the current paper, we will briefly outline the process.

The next step in the MSC processing is to convert the MIMD program into a conventional state machine. This is done by converting the code into a control flow graph in which each node, or original state, represents a basic block [CoS70]. We assume that these basic blocks are made maximal by a combination of simple local optimizations, removal

```
if (nanocontroller_is_0) { program_0 }
else if (nanocontroller_is_1) { program_1 }
else if (nanocontroller_is_2) { program_2 }
...
else if (nanocontroller_is_last) { program_last }
```

Listing 1: Creation Of MIMD-Parallel Program

of empty nodes, and code straightening [CoS70]. However, in order to represent *arbitrary* global and interprocedural control flow, a few tricks are needed.

One might expect that supporting arbitrary global control flow would be a problem, but it is not. In practice, it is most common that each state will have zero, one, or two exit arcs. Zero exit arcs mark termination of the program — for example, a block ending with **`exit(0)`** in a unix C program. Two arcs most often correspond to the **`then`** and **`else`** clauses of an **`if`** statement or to the exit and continuation of a loop. However, constructs like C's **`switch`** or Fortran's computed-**`goto`** can yield more than two exit arcs. For MSC, there is no algorithmic restriction on the number of control flow exit arcs a state may have. Of course, state machines representing loops, even irreducible ones, also are perfectly valid. Likewise, function calls are treated precisely as **`goto`**s, even if they are recursive; **`return`** statements are treated as computed-**`goto`**s that select among the possible return addresses.

```
if (A) {
    do { B } while (C);
} else {
    do { D } while (E);
}
F
```

Listing 2: Simple Example

Listing 2 gives C code for a simple example taken from the MIMD-to-SIMD MSC paper [DiK93]. The result of mechanically constructing the MIMD state graph is given in Figure 2.

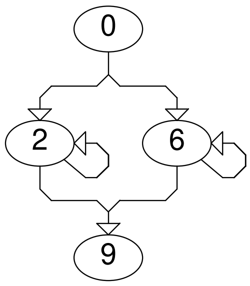


Figure 2: MIMD State Graph for Simple Example

The basic MSC algorithm then applies a process that looks remarkably like that used in constructing lexical analyzers by converting a Nondeterministic Finite Automaton (NFA) into a Deterministic Finite Automaton (DFA). The only difference is in the definition of *reaching* a state: SIMD meta-states are created for every possible combination of next states. Thus, basic MSC converts Figure 2 into Figure 3.

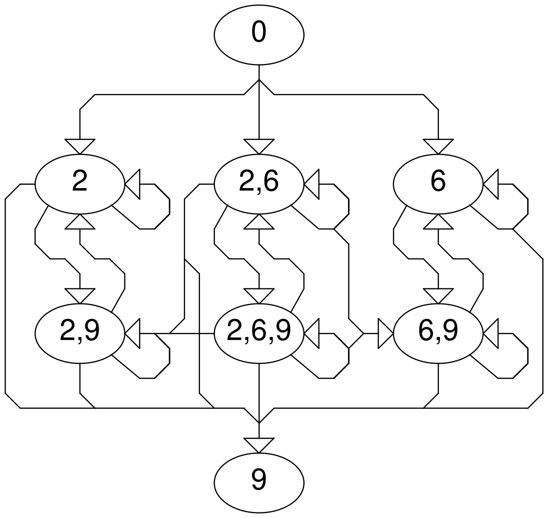


Figure 3: Basic Meta-State Graph for Simple Example

For example, the node labeled with *2,6* contains the code from *both* original state 2 and 6. In SIMD execution, nanocontrollers that were logically in MIMD state 2 would *disable themselves* while instructions from 6 were broadcast; nanocontrollers in state 6 would do exactly the opposite. If there are any instructions that appear in both original state 2 and 6, they can be factored-out so that those instructions are broadcast by the SIMD control unit once and executed by both nanocontrollers in MIMD state 2 and those in MIMD state 6. A supplementary compiler transformation called Common Subexpression Induction (CSI) [Die92] is applied to restructure the code within each meta-state to maximize the sharing of instructions... with a SIMD instruction set designed to facilitate instruction matching, CSI is very effective.

When the end of state *2,6* is reached, there are no fewer than 5 possible next meta-states: *2,6*, *2,6,9*, *2,9*, *6,9*, and *9*. Determining which meta-state the SIMD control unit should broadcast next is done by a global voting process. In essence, the SIMD control unit collects a 3-bit *globalor* of the next state choices for all nanocontrollers; these bits respectively answer the questions:

- Does any nanocontroller want to enter MIMD state 2?

- Does any nanocontroller want to enter MIMD state 6?
- Does any nanocontroller want to enter MIMD state 9?

Given this vote, the SIMD control unit simply executes a computed-`goto` to jump to the appropriate next meta state based on the 3-bit value collected. The register-allocation-like problem alluded to earlier arises if the set of MIMD states following a particular SIMD meta-state is larger than the number of bits that can be directly managed; reassignment of bits solves the problem.

Unfortunately, the meta-state graphs (e.g., Figure 3) can have significantly more nodes than the original MIMD state graphs (e.g., Figure 2). As with NFA to DFA conversion, meta-state conversion can yield an exponential increase in the number of states! It could be argued that, given SIMD instructions stored in off-chip RAM, the increase in code size is not important. However, it is possible to dramatically decrease the number of meta states by controlled merging into *covering states*. An extreme version of this merging is shown in Figure 4; the only penalty is potentially less-effective CSI.

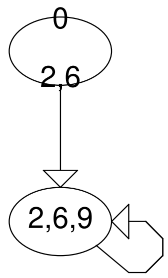


Figure 4: Merge-Compressed Meta-State Graph

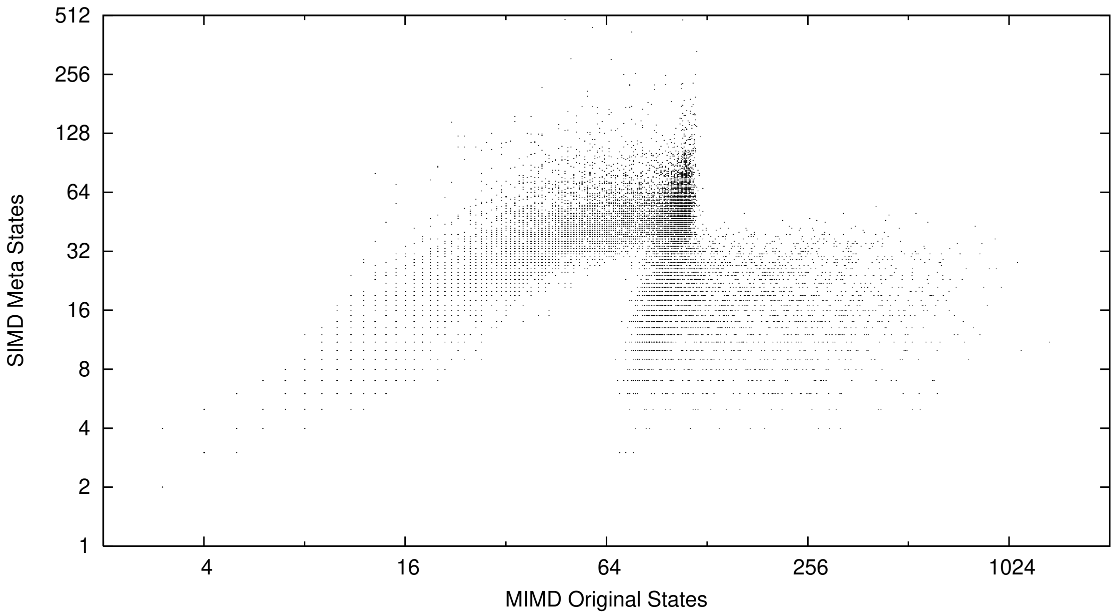
4.3. Does MSC Have Predictable Execution Times?

The final MSC complication relates back to requirement 2.2: predictable execution timing behavior. There are really two types of timing behavior: one whose solution was given in [DiK93], the other is a new problem arising in the real-time nature of nanocontrollers.

the first type of timing is relative timing of MIMD states being executed: is progress made fairly by all nanocontrollers? This property can be preserved arbitrarily accurately, but doing so requires *cracking* states into smaller chunks to ensure more equal progress is made. For example, if state *2* took twice as long to execute as state *6*, we might first crack state *2* into the state sequence *2a* and *2b*, then create a meta state for *2a,6*. Taken to extremes, this cracking can greatly magnify the number of meta states generated.

As a test, over 30,000 randomly-generated MIMD programs involving **`if`**, **`while`**, and operations on 8 registers were processed by the MSC algorithm with a time-splitting rule that required no nanocontroller to make more than twice as much progress within any given meta state as any other nanocontroller in that meta state. The results are plotted as a scatter graph in Figure 5. The artifacts around 128 MIMD states are probably caused by language-construct nesting-depth limits within the MSC's C-dialect parser, which rejected certain randomly-generated programs. Despite that defect, this level of time-splitting for fairness clearly does not result in explosive growth in the number of meta states.

Figre 5: MIMD Original States vs SIMD Meta States



The other aspect of predictable timing relates to the fact that nanocontroller programs need to interact with the devices they control with precise real-time constraints. It is theoretically possible to preserve precise instruction-level timing for specific instructions as the MSC process is performed, but the complexity of doing so makes that solution impractical. What is needed is a simple way to ensure specific instructions execute at particular times.

The key to a simple solution is in our description of requirement 2.2: the level at which real-time constraints must be met is expected to be several orders of magnitude larger than the time taken to execute an instruction. For example, real-time resolution might be a microsecond, with an instruction executed every nanosecond. By *reserving* a small fraction of SIMD instruction broadcast slots for precisely timed operations, normal execution is virtually unaffected but cycle-accurate timing can be ensured. Given that the precisely-timed operations always will involve I/O, it is very likely that all nanocontrollers needing input can share a single reserved input cycle and all nanocontrollers needing output can share a single reserved output cycle.

The original MSC algorithm did not have any provision for reserving SIMD instruction slots, but it did include a method for converting synchronizations into meta state construction constraints. An I/O reservation is nothing more than synchronization with an “imaginary” nanocontroller whose sole purpose is to participate in a barrier synchronization with every tick of the real-time clock.

5. Architecture Revisited

Given the large body of work on design of SIMD systems, it is not necessary to present an overview. Neither would it be appropriate to present a detailed design at this time because the details of the nanocontroller design should be tuned to the particular application. However, having reviewed the basic properties of MSC, it is now appropriate to discuss how basic SIMD architecture can be modified to better match MSC-driven nanocontroller applications in general.

Every SIMD processing element needs some kind of Arithmetic/Logic Unit (ALU). The ALU does not need to directly perform a wide range of operations within a single clock cycle, nor does it need to work on more than 1 bit per value, but over multiple clock cycles it must be capable of performing a complete set of 2-input bitwise logic functions (e.g., NAND would suffice). The fundamental building block of modern computer arithmetic is 2's complement binary addition, which at the bit level is actually a function of three inputs generating two outputs: given two data values and a value carried from less significant bit positions, compute the resulting sum and carry outputs. Simulating a three-input, two-output, function using single-output two-input logic takes multiple steps, so virtually every SIMD design incorporates a 3-input "Full Adder" circuit or a generalization thereof (e.g., the CM2 uses two 8-input multiplexors [TMC89]).

There must be some amount of local data storage. How much is appropriate will vary widely depending on the application, but most nanocontroller applications probably will need fewer than 100 bits in registers local to each processing element. Keep in mind that data held by other processors or by the SIMD control logic also can be accessed (more slowly), so data space is not really limited to that which is kept local but is multiplied by the number of processing elements. It also is possible to trade *data space* for *control space*, and control space is not replicated per SIMD processing element. For example, storing the value 8 can be accomplished either by storing the value 8 in a local register or by jumping into a state in which 8 is *assumed* to be the value. The SIMD meta-state program may get very large and processing somewhat inefficient, but the theoretical minimum number of bits needed local to a processing element is literally just enough to hold a unique identifier (e.g., enough to hold a processing element number).

To increase the efficiency of CSI within MSC, there are a few important performance tweaks we can make to both ALU and local register file/memory designs. Recall that CSI attempts to recognize (or coerce) multiple different program fragments to have as many instructions in common as possible; thus, CSI is very sensitive to how operations and data references are encoded in instructions. Generally, CSI is more effective when:

- The commonly-used set of instruction bit patterns is small. If there are few bit patterns used, repeats in different code segments are more likely. Encoding multi-bit constants in instructions makes CSI less effective; this is true whether the constants are immediate values, register numbers or memory addresses, or opcode options.
- Modal instruction encoding is used. If an ALU operation is set with each instruction, then each step of an Add is different from each step of an Or. Alternatively, if the Add operation is set in some processing elements and Or in others, then the bitwise repeats of both operations can use the same instructions.
- Indirect addressing is used. For example, a stack Add instruction pairs with another stack Add even if the stacks in the processing elements contain different numbers of elements, whereas Add r1,r2 would not pair with Add r1,r3. A stack is just one example of indirect addressing; the general concept is to locally modify the broadcast reference. Especially for bitwise ALUs, it may be worthwhile to use shift registers to shift data past ALU I/O connections rather than to use decoding logic to address the desired cell.

The last aspect of SIMD design to discuss is the Control Unit (CU). There are just two primary issues involving design of the CU:

- In constructing SIMD supercomputers that span multiple chips, the CU broadcast speed often limited the maximum clock rate for the processing elements; a similar phenomenon now occurs within a single chip containing many processing elements. The result is that rather than a single CU, a hierarchy of CUs should be used to distribute ever-finer-grain control information. Only the sub-CUs nearest the processing elements should be broadcasting at the full clock rate.
- The CU should employ sophisticated block-oriented optimizations in accessing off-chip RAM. For example, each block could be tagged with CU-only instructions that explicitly managed prefetch and caching of future blocks. An additional complication is that the caching must be such that no time-critical operations can be delayed by a cache miss; thus, *reserved* instruction slots should be handled separately if a

disturbance from a cache miss would otherwise be possible.

Finally, it is widely known that bit-serial processing elements often can achieve better performance by making use of bit-level optimizations rather than blindly expanding word-level operations into fixed bit-level translations. A similar benefit can be obtained by performing CSI (and perhaps MSC) on bit-level instructions.

6. Conclusions and Future Work

This paper introduced a vision of nanocontrollers as very low circuit complexity controllers that are fully programmable. It was suggested that SIMD-based architectures, combined with Meta-State Conversion (MSC) compiler technology, could provide a viable implementation method, and a detailed overview of the approach was given.

As a “straw man” design, consider using MSC to support a full MIMD, C-dialect, nanocontroller programming model using an architecture based on the Thinking Machines CM2 [TMC89]. With data storage per processing element reduced to 64 bits, circuit complexity per nanocontroller should be well under 500 transistors. Current technology would allow a clock rate of at least 1GHz with 3 clock cycles/bit operated on. These numbers do not quite reach our goals for a nanocontroller, but they are several orders of magnitude closer than traditional approaches.

Aside from working toward further development of nanocontrollers along the lines described in this paper, it is interesting to note that *analog* technology can be directly useful in building nanocontrollers. For example, the carry logic of a Full Adder can easily be replaced by a single *threshold logic* gate — after all, the carry output is simply the answer to “is the sum greater than 1?” More aggressive use of analog elements may be the way to achieve one more order of magnitude reduction in circuit complexity.

References

[Abu97] Nael B. Abu-Ghazaleh, *Shared Control Multiprocessors - A Paradigm for Supporting Control Parallelism on SIMD-like Architectures*, PhD Dissertation, University of Cincinnati, July 1997.

[Bat74] K. Batcher, “STARAN Parallel Processor System Hardware,” *Proc. of the 1974 National Computer Conference,* AFIPS Conference Proceedings, vol. 43, pp. 405-410.

[Bat80] K. Batcher, “Architecture of a Massively Parallel Processor,” *Proc. of IEEE/ACM International Conference on Computer Architecture*, 1980, pp. 168-173.

[Bla90] T. Blank, “The MasPar MP-1 Architecture,” 35th IEEE Computer Society International Conference (COMPCON), February 1990, pp. 20-24.

[CoS70] J. Cocke and J.T. Schwartz, *Programming Languages and Their Compilers,* Courant Institute of Mathematical Sciences, New York University, April 1970.

[DiC92] H.G. Dietz and W.E. Cohen, “A Control-Parallel Programming Model Implemented On SIMD Hardware,” in Proceedings of the *Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[Die92] H.G. Dietz, “Common Subexpression Induction,” Proceedings of the *1992 International Conference on Parallel Processing,* Saint Charles, Illinois, August 1992, vol. II, pp. 174-182.

[DiK93] H. G. Dietz and G. Krishnamurthy, “Meta-State Conversion,” *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 47-56, Saint Charles, Illinois, August 1993.

[DiZ91] H.G. Dietz, M.T. O’Keefe, and A. Zaafrani, “An Introduction to Static Scheduling for MIMD Architectures,” Advances in Languages and Compilers for Parallel Processing, edited by A. Nicolau, D. Gelernter, T. Gross, and D. Padua, The MIT Press, Cambridge, Massachusetts, 1991, pp. 425-444.

[NiT90] M. Nilsson and H. Tanaka, "MIMD Execution by SIMD Computers," Journal of Information Processing, Information Processing Society of Japan, vol. 13, no. 1, 1990, pp. 58-61.

[Par95] Behrooz Parhami, "*SIMD Machines: Do They Have A Significant Future?* Report on a Panel Discussion," IEEE Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean, VA, Feb. 6-9, 1995.

[TMC89] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, Version 5.1, May 1989.

[WiH91] P.A. Wilsey, D.A. Hensgen, C.E. Slusher, N.B. Abu-Ghazaleh, and D.Y. Hollinden, "Exploiting SIMD Computers for Mutant Program Execution," Technical Report No. TR 133-11-91, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio, November 1991.