SNAP: A Sensor-Network Asynchronous Processor

Clinton Kelly, IV; Virantha Ekanayake; Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca NY 14853, U.S.A.

Abstract

We present a Sensor-Network Asynchronous Processor (SNAP), which we have designed to be both a processor core for a sensor-network node and a component of a chip multiprocessor, the Network on a Chip (NoC), which will execute a novel sensor-network simulator. We discuss the advantages of using the same processor for nodes in physical and simulated sensor networks. We describe the attributes that a processor must possess to function well in both roles, and we then describe the way we designed SNAP to have these attributes.

1 Introduction

Sensor networks are typically comprised of many low-cost nodes that can be used to gather, process, and propagate a wide variety of information from the surrounding environment. Recently, interest has focused on self-configuring *wireless* sensor networks and the unique challenges they pose, such as managing dynamic network topologies and maximizing the lifetime of networks in the context of limited sensor-node energy budgets [27].

Most of the application development and communication-protocol design for these sensor nodes is done using network simulators such as `ns-2` [22] and Glomosim [36]. After the application and protocol software functions properly in the simulation environment, it is then deployed on the actual nodes, each of which contains at the very least a processing element, a radio interface, and some way of interacting with its environment. Today's sensor nodes typically use commodity microcontrollers for their processing elements: For example, the Berkeley TinyOS Motes [14] and UCLA's MEDUSA-II sensors [34] use Atmel's low-cost 8-bit RISC AVR series of microcontrollers, while a more high-end sensor from Rockwell [35] uses Intel's StrongARM SA1100 32-bit controller. Unfortunately, the behavior predicted by simulation may vary dramatically from that observed in the real network [15]; researchers must typically perform several debug-and-test cycles before the sensor network actually performs as predicted, and even this has only been achieved for small networks.

Much of the complexity of deploying wireless sensor networks arises due to the disparity between the simulation and the actual hardware implementation. Even the most detailed simulation models used in `ns-2` and Glomosim do not accurately model the hardware limitations—such as limited message buffering, memory allocation latencies, or processing-time requirements—of the actual node on which the sensor application will run [4]. Moreover, modifying `ns-2` or Glomosim to accurately model these factors would probably not be useful, as the time required to simulate several hundred to several-thousand nodes would become unreasonably long (and would require a great deal of memory).

With this in mind, we are developing an integrated hardware simulation-and-deployment platform for wireless sensor networks, based on a 16-bit message-passing asynchronous processor called the Sensor-Network Asynchronous Processor (SNAP). In this paper, we present the design and architecture of SNAP, which will be used for two main purposes: (1) to be the main processor for a sensor-network node that we are designing at Cornell and (2) to be a component of the Network on a Chip (NoC), a custom chip multiprocessor designed for sensor-network simulation [10, 16].

The creation of SNAP will have several benefits for researchers studying sensor networks:

Faster simulation of large-scale sensor networks. SNAP is the main building block of the NoC, which will be able to efficiently execute parallel network simulators that use a specific synchronization protocol, called *time-based synchronization* (TBS). (We describe TBS briefly in Section 2.) A simulator using this protocol will be able to simulate large-scale sensor networks (i.e. those containing on the order of 100,000 nodes) faster than real time and many orders of magnitude faster than software simulators [10] . Fast simulation of such networks will be an invaluable tool for researchers.

A common software interface. Because we will use SNAP to build both physical *and* simulated nodes, researchers studying sensor networks will be able to use a single software interface: A researcher will be able to use the same protocol code for the simulation and implementation of a given node in a sensor network.

Evaluation of network simulators. The SNAPs in physical-network nodes and the SNAPs that simulate the physical-network nodes will execute almost-identical programs (the SNAPs in the real nodes will not need to execute code to simulate radio and channel layers, because they will have real radios). Therefore, researchers will be able to easily evaluate the realism of simulated radio and channel layers by comparing the behavior of real sensor networks with those simulated on the NoC. This evaluation of radio and channel layers will lead to more accurate network simulations; studies have shown that the results of such simulations are very sensitive to the parameters used for physical layer modeling [30].

The rest of this paper reads as follows: In Section 2 we briefly describe TBS and the NoC, and discuss the attributes that SNAP should posses if it is to execute a TBS-based simulator. Section 3 describes how we designed SNAP to have these attributes. In Section 4 we describe the implementation details of some of the more interesting components of SNAP, such as the instruction-fetch unit and the timer coprocessor. Section 5 predicts the cycle-time of our processor and summarizes early simulation results regarding the scalability of TBS-based sensor-network simulations running on the NoC. In Section 6 we discuss related work in network simulation, multicomputing, and sensor networks. We summarize our work in Section 7.

2 TBS and the NoC

Most network simulators, including the TBS-based simulator for which we designed the NoC, are *discrete-event simulators.* A discrete-event simulator proceeds by constantly removing the timestamped event from the head of its time-ordered event queue and then simulating the effects of the event (we call simulating the effects of an event "executing" an event). High-performance simulators use a technique called parallel discrete-event simulation (PDES). Fujimoto [6] provides a thorough discussion of PDES. Processors in such a parallel simulation can schedule events in one another's queues by passing *messages,* which contain events. The processors use a *synchronization protocol* to ensure that they always execute all events, including those received from other processors, in nondecreasing timestamp order.

Our synchronization protocol, TBS, is unique in that it requires every processor to execute events at scaled versions of the events' timestamps. In other words, if a processor has an event E with a timestamp T in its event queue, it will not execute E until $T \geq s \times t$, where t is the real time elapsed since the simulation began, and s is called the simulation's *time scale.* An event with a timestamp that satisfies this inequality is said to be *executable.* If s is greater than one, then the simulation proceeds faster than real time.

In a TBS-based sensor-network simulation running on the NoC, each SNAP simulates one node in the sensor network (this is different from traditional simulators, in which one processor typically simulates a large number of nodes [36]). We estimate that a single NoC chip will contain 100 SNAPs. These processors can pass messages amongst themselves via a pipelined, asynchronous interconnect [10]. Because researchers often desire to simulate sensor networks containing as many as 100,000 nodes, we have designed the NoC such that we can gluelessly assemble multiple chips together for such large simulations. An off-chip workstation, called the *host*, controls simulations by exchanging messages with the SNAPs in the NoC. The host sends messages to the SNAPs containing information such as the code the SNAPs should run for a given simulation, directions telling the SNAPs when to start and stop simulations, and simulated sensor data. The host receives messages from the SNAPs containing statistics, error information, etc.

In [10], we show that a sufficient condition for correctness in a TBS-based simulation is that every message, containing an event with timestamp T, must arrive at its destination SNAP while $T < s \times t$ (so that the destination SNAP may insert the enclosed event into its event queue before the event becomes executable). Because of the real-time nature of TBS, a processor that can correctly simulate, with a time scale greater than one, one node in a TBS-based simulation of a sensor network can also execute the code for a node in a real sensor network by using a time scale equal to one (and replacing the code that simulates the radio layer with a physical radio).

We require two characteristics of SNAP if it is to

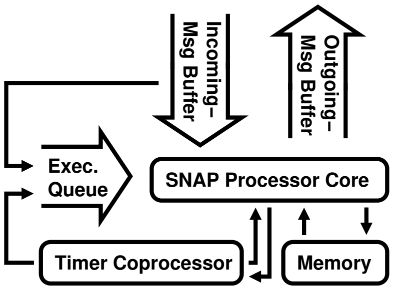

Figure 1. A high-level depiction of SNAP.

efficiently execute a TBS-based simulation (and therefore also act as a main processor for an actual sensor-network node): (1) the ability to quickly exchange messages with other SNAPs in the NoC (lower latency for message-passing between processors directly affects the time scale [10]) and (2) the ability to easily manage its event queue ("managing" an event queue includes scheduling events, canceling previously-scheduled events, and determining when events have become executable). We would also like the area of a single SNAP to be as small as possible, so that we can reach our estimate of 100 processors per chip; although we have designed the NoC such that we can simulate large networks by connecting several chips (we will connect these chips using the mechanism described in [33], we would still like a single chip to contain many SNAP cores. Moreover, SNAP's small size will be useful when we use it as a component of a physical sensor-network node, which will ideally be as small as possible. In the next section, we discuss the architecture of the NoC, and describe how we have designed it to meet our three main goals: fast message passing, easy event-queue management, and low area.

3 The SNAP Architecture

Figure 1 depicts the components of SNAP. The *processor core* receives messages from the NoC interconnect (or from the sensor-network node's radio) via the *incoming-message buffer* and sends messages to the NoC interconnect (or the radio) via the *outgoing-message buffer.* The *executable queue* contains tokens that represent executable events or newly arrived messages. Tokens can be inserted into this queue by the incoming-message buffer or by the *timer coprocessor*, which the processor core uses to manage its event queue. The processor core reads from and writes to the main memory.

In this section we provide a high-level overview of SNAP. We organize this overview by explaining how the three requirements from the previous section influence SNAP's design. We begin by discussing the aspects of the design that were chosen to minimize area.

Note that the following discussion describes the version of SNAP that will be used as a part of the NoC. At the end of the section we will describe the differences between this version and the version that will be used in the sensor-network node.

3.1 Minimizing Area

Aspects of our design that were chosen to minimize area include the following:

- SNAP lacks a cache. The memory requirements for simulating a single sensor-network node are modest enough that each SNAP's memory will be fairly small, enabling single-cycle reads and writes even without a cache.
- The processor core does not support virtual memory or exceptions.
- The processor core does not have a multiply/divide unit or a floating-point unit.
- The processor core has a 16-bit data path. Instructions can be either 16-bit or 32-bit (single- or double-word).
- SNAP's memory is made from asynchronous DRAM (750 λ^2/bit[5]), instead of faster-but-less-dense SRAM (1200λ^2/bit) [25].

The use of variable-length instructions makes the instruction-fetch loop interesting; we shall discuss this in Section 4.1.

When SNAP boots, the processor core executes a short sequence of boot code. This code directs the core to place the first incoming message, called its *startup message*, into a certain address in memory, and then to jump to this address. (The startup message will presumably come from the host, but could also come from another SNAP.) The startup message contains all of the code for the entire sensor-network simulation, including some initialization code. The last instruction in this initialization code will be the `DONE` instruction, which is the first of the event-manipulation instructions that we will consider.

3.2 Manipulating the Event Queue

The software running on SNAP issues the `DONE` instruction when it has finished executing one of the following: its startup message, an event, or the code that

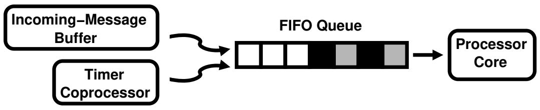

Figure 2. The executable queue. The incoming-message buffer and the timer coprocessor insert tokens into a FIFO queue, called the *executable queue.* These tokens tell the processor to which address it should jump after issuing a `DONE` instruction.

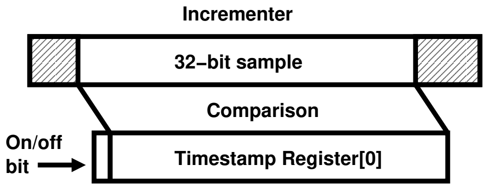

Figure 3. The incrementer and a single timestamp register.

schedules the event that an incoming message contains. The `DONE` instruction tells the processor to wait until one of two things happens:

- When *a previously-scheduled event becomes executable,* the processor core jumps to a specific address that contains the code to execute the event. When the core finishes executing the event, the software should issue another `DONE` instruction.

- When *a new message arrives from the interconnect,* the processor core jumps to a specific address that contains the code to schedule the enclosed event. When the core finishes scheduling the event, the software should issue another `DONE` instruction.

The processor decides what code to execute after a `DONE` instruction by reading tokens from the executable queue (see Figure 2). Because the `DONE` instruction can cause the processor to stall (if the executable queue is empty), it is somewhat similar to the `HALT` instruction in the Amulet3i [7].

The timer coprocessor is a hardware implementation of the event queue. It contains a set of seven 32-bit *timestamp registers* and a 40-bit *incrementer.* A timestamp register can be either "on" or "off." Every timestamp register that is "on" corresponds to an event in the event queue, and contains the timestamp of that event. A 32-bit, contiguous sample of the incrementer corresponds to the scaled version of the current time. Whenever the value in a timestamp register is equal to the sample of the incrementer, the event corresponding to the timestamp register is executable: The timer coprocessor inserts a token representing the event into the executable queue (the token itself contains only a number, which identifies the appropriate timestamp register), and changes the timestamp register to "off." See Figure 3. When the processor core removes a token from this queue, it reads the appropriate address from a table (the table has eight rows: one for every timestamp register, and one for incoming messages) and sets the program counter to that address. SNAP's Instruction Set Architecture (ISA) includes instructions for changing this table. Section 4.3 discusses the implementation of the timer coprocessor.

The software running on SNAP can adjust the time scale of the simulation by taking different samples of the incrementer: Shifting the sample one bit to the left corresponds to doubling the time scale. To schedule an event, the software uses an instruction to set a timestamp register to the timestamp of the event it wishes to schedule. The ISA also includes instructions that turn off timestamp registers (cancel events), and shift the sample of the incrementer.

3.3 Sending and Receiving Messages

SNAP sends and receives messages via its outgoing-message buffer and incoming-message buffer, respectively. Rather than adding extra instructions for sending and receiving messages, we use a scheme similar to that used in the Mosaic Element [13] and in the Raw microprocessor [31]: Of the sixteen general-purpose registers in the processor, one, register zero, is always zero, and another, register fifteen (r15), maps to the message queues. The processor core places any value to be written to r15 into the outgoing buffer, and whenever an instruction reads r15, the processor core removes the value at the head of the incoming buffer and uses it as the operand in the instruction. For example, when the processor core executes the instruction `ADD r15, r5, r15`, it removes the value at the head of the incoming-message queue, adds that value to the value of `r5`, and sends the result into the outgoing-message queue (SNAP will block until a value is available at the head of the incoming-message queue).

If the NoC interconnect is congested, an instruction writing to r15 (the outgoing-message buffer) will fail—the processor will skip the instruction. A program can avoid the processor skipping any instructions by checking the outgoing-message buffer's *status register*, which contains the number of free spaces remaining in the outgoing buffer. Programs that send long messages should check the status register to make sure that there is enough space in the buffer for the entire message. We designed this scheme to prevent deadlock in the NoC:

The software can detect when the network is busy, and therefore prevent itself from being blocked on a send operation, thereby preventing deadlock.

3.4 SNAP in a Sensor Network

We will now briefly describe the changes necessary to use SNAP as a part of a real sensor-network node, as opposed to a part of the NoC. The main change is that the incoming- and outgoing-message buffers will be connected to a real antenna, instead of to the NoC interconnect. Because this antenna will never be "congested," this version of SNAP will not need a status register. The presence of a real radio will also require several new types of tokens for the executable queue. The radio interface should place tokens in the queue whenever certain radio "events" occur, such as the radio changing modes (radio modes include "transmitting," "sensing," "receiving," etc.), the radio receiving a packet, etc.

The actual interface between SNAP and the radio transceiver will consist of a low-speed serial input/output pair of lines, a two-bit control interface to select the receive/transmit mode, and two lines for the send and receive clock. Because the off-chip data rates are low (in the range of 115kbps[26]), there will be plenty of time to synchronize the sending and receiving of data with the external clock. Incoming and outgoing messages will be translated to the off-chip serial format required by standard RF transceivers [26].

4 Implementation Details

SNAP is composed of quasi-delay-insensitive (QDI) asynchronous circuits [17]. We chose to use asynchronous circuits because sensor-network nodes typically have tight energy budgets. Moreover, the design flow of QDI asynchronous circuits makes them a natural choice for implementing a parallel network simulator. In this design flow, a circuit is first specified as a collection of sequential processes that communicate via message passing. Likewise, a parallel network simulator can be described as a collection of "logical processes" that communicate by passing messages containing timestamped events [6].

When designing SNAP, we combined techniques used in the designs of the Caltech Asynchronous Microprocessor [20] and in more pipelined processors such as the MiniMIPS [21] and ASPRO-216 [28]. The Caltech processor is relatively unpipelined and uses communication actions on probed channels to synchronize access to shared variables; it takes up little area and is low-power, but is not very fast. The MiniMIPS and ASPRO use finely-pipelined circuits to obtain high throughput; they are less area- and energy-efficient, but they have much lower cycle times. We chose to mix the two styles because we wanted SNAP to have the area- and energy-efficiency of the Caltech Asynchronous Microprocessor while still displaying some of the throughput advantages of the MiniMIPS and ASPRO. To increase throughput, we also employed the techniques described in [18] for pipelined mutual exclusion.

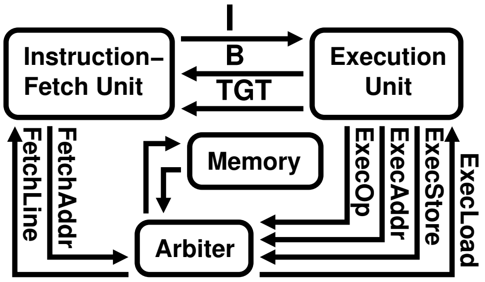

Figure 4. The instruction-fetch unit and the execution unit.

SNAP's processor core consists of two parts: an instruction-fetch unit and an execution unit. The former fetches instructions and updates the program counter, while the latter decodes and executes instructions. Within each unit, we have used a control-data decomposition design style in which the datapath is synchronized by dataless communications on slackless, probed channels [19]. The units communicate with each other via highly-pipelined data channels; the high degree of slack on these channels allows the two units to operate concurrently. Figure 4 depicts these data channels. The instruction-fetch unit uses the I channel to send instructions or the value of the program counter to the execution unit. The execution unit uses the B channel to send branch information to the instruction-fetch unit, and, in the case of a jump-register instruction, uses the TGT channel to send the target address of the jump. Figure 4 also shows the arbiter that controls access to the first bank of main memory (which contains data and instructions).

In this section, we discuss the implementation of the instruction-fetch and execution units. We also describe the timer coprocessor, and provide an overview of SNAP's memory system.

4.1 The Instruction-Fetch Unit

SNAP's instruction-fetch unit performs two operations: it updates the program counter, and it uses the new value of the program counter to fetch a line from memory. One of our main goals in designing SNAP was to develop an instruction-fetch loop that would be efficient, despite our choice to support variable-length instructions. To achieve this goal, we used a technique similar to that of branch-delay slots: In a processor that uses single-word instructions and has a one-cycle branch-delay slot, consider that every instruction determines how the processor will update the program counter after *the next instruction.* For example, a jump-register instruction in the MIPS ISA [9] indicates that, after executing the instruction in the branch-delay slot, the processor will change its program counter to a register value. Similarly, any arithmetic instruction indicates that, after executing the next instruction, the processor will increment the program counter by four.

SNAP uses a similar scheme: Rather than having each instruction indicate how to update the program counter after fetching the next instruction, each *word* indicates how to update the program counter after fetching the next *word.* The execution unit performs all decoding and uses the B channel to send tokens to the instruction-fetch unit specifying how to perform this update. For example, consider how the SNAP processor core will execute an `ADDI` instruction, which uses two words (the first specifies the opcode and identifies two registers, the second is an immediate). After receiving the first word, the execution unit will send two tokens on the B channel, both telling the instruction-fetch unit to increment the program counter by one. (The execution unit can send both tokens immediately because the B channel is pipelined.) As a second example, consider how the core will execute the `JUMP` instruction, which also consists of two words, the first containing the opcode, and the second containing the destination of the jump. After receiving the first word, the execution unit will send two tokens on the B channel: The first will tell the instruction-fetch unit to update the program counter with the value of the next word, and the second to increment the program counter by one. (The instruction-fetch unit does not send the second word to the execution unit, since it will use this word to update the program counter.) Because each word determines how to update the program counter after fetching *the next word,* the instruction-fetch unit can fetch one word while the execution unit decodes and executes the previous word.

Figure 5 shows the instruction-fetch unit in detail; the dashed lines represent control channels, and the solid lines represent data channels. The *Control* process receives from the B channel tokens that tell it how to update pc. The pc can be updated with the value from an incrementer (for a non-branch instruction) or

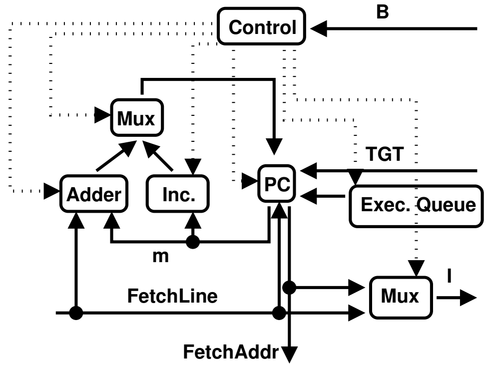

Figure 5. The instruction-fetch unit. Dashed lines indicate control channels; solid lines indicate data channels.

an adder (for a relative branch), from a token taken from the TGT channel (for a jump-register instruction) or the *FetchLine* channel (for an absolute jump), or from the executable queue (for a `DONE` instruction).

Although there are several instructions in the SNAP ISA that require the instruction-fetch unit to perform operations other than incrementing the program counter, the only one of these that is unique to our processor is the `DONE` instruction. When the execution unit receives the `DONE` instruction, it sends a token to the instruction-fetch unit telling it to update the program counter based on the value at the head of the executable queue.

4.2 The Execution Unit

The execution unit contains decode logic, function blocks, and the register file. The function blocks include an adder, a shifter, a logic block, a bit-field block (SNAP's ISA includes bit-field-set and bit-field-read instructions [10]), a memory-interface block (which calculates addresses), a timer-coprocessor block (which controls scheduling and canceling events), and a conditional-branch block. An ISA summary is given in the Appendix. Figure 6 shows the execution unit.

Because its implementation of the execution unit is very similar to that of the Caltech Asynchronous Microprocessor's execution unit, we do not discuss it in great detail here. The main difference between our execution unit and the corresponding circuits in the Caltech Asynchronous Microprocessor is that our execution unit uses pipelined mutual exclusion [18]. This technique uses shared lock variables to pipeline control distribution while preserving mutual exclusion; moreover, it can be used to implement mutual exclusion conditions that are conjunctions of pairwise mutual exclusion constraints among processes. Pipelined mutual

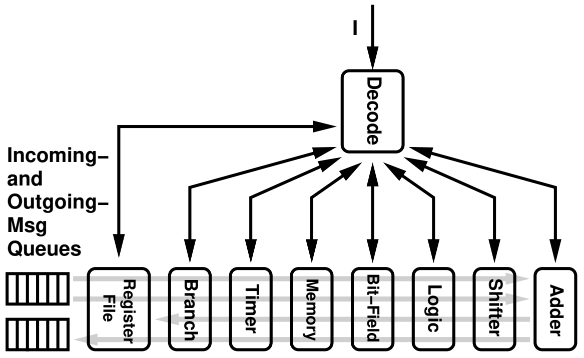

Figure 6. The execution unit. Busses are shown in gray. Note that the two read busses and one of the write busses connect to the incoming- and outgoing-message buffers, respectively.

exclusion is similar to the register locking used in the Amulet 1 [24] but more general in terms of conditions and amount of decoupling permitted, and results in QDI circuits. These techniques allow for greater parallelism between different function blocks than was possible in the Caltech Asynchronous Microprocessor.

4.3 The Timer Coprocessor

The timer coprocessor contains a self-incrementing counter, which holds a scaled version of real time. The rest of the coprocessor must be fast enough to keep up with the rate at which the incrementer changes: Every timestamp register must be compared against the incrementer every time the incrementer's value changes. In this section, we describe how we have designed the coprocessor to meet this throughput requirement.

Although the timer coprocessor must compare every timestamp register against the value of the incrementer every time the incrementer changes, it does not have to *finish* the comparisons before the incrementer changes again: We can pipeline the comparison process such that one set of comparisons (the comparisons between every timestamp register and one value of the incrementer) completes every cycle. Figure 7 depicts a portion of the incrementer and a single timestamp register. The coprocessor is made up mostly of multiple instantiations of three basic circuits:

1. The *incrementer processes* are essentially the same as those of a normal incrementer, except that they send copies of their values to the shift units after every cycle. These processes propagate high or low carry-out values.

2. The *shift units* route the values of the incrementer processes to the values of the corresponding timestamp-register processes (TSRPs). A shift unit determines to which TSRP to forward values based on the time scale used for the simulation. Shift units only forward values to TSRPs that correspond to timestamp registers that are "on."

3. The TSRPs contain the values of the timestamp registers. Every cycle, each TSRP compares its value against the value it received from its corresponding shift unit.

The incrementer processes and the TSRPs contain 2-bits per process, using a 1-of-4 code.

A timestamp register is equal to the value of the incrementer if each TSRP's value is equal to the value it receives from its shift unit. This logical AND computation ripples from the lowest-order to the highest-order TSRP: Each TSRP ANDs the result of its comparison with a boolean value it receives from the TSRP on its right, and forwards this result to the TSRP on its left. If the value forwarded by the highest-order TSRP is true, the timer coprocessor inserts a token into the executable queue, and sets the appropriate timestamp register to "off."

Figure 7 shows the TSRPs that correspond to only a portion of a single timestamp register. In the actual timer coprocessor, each shift unit may make as many as seven copies of the value from the corresponding incrementer process. Figure 7 also omits the channels that allow the processor core to write the values of the TSRPs, and turn timestamp registers on and off.

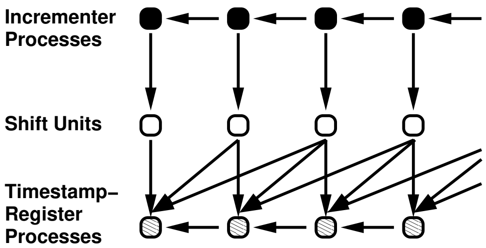

Figure 7. A portion of the timer coprocessor. The coprocessor is pipelined horizontally: Every time the lowest-order-bit incrementer process receives a carry-in signal, the highest-order-bit timestamp-register process sends a boolean value.

The lowest-order-bit incrementer process (LOBIP) receives its carry-in signal from a synchronous process, with which it completes a full QDI handshake. To the LOBIP, this synchronous process is indistinguishable from another incrementer process, as long as the LOBIP finishes the handshake before synchronous circuit begins the next increment operation. The entire timer coprocessor is highly-pipelined so that it can achieve throughput high enough to enable one comparison to complete every time the incrementer changes its value.

4.4 The Memory System

Although SNAP has only one memory, which contains instructions and data, its memory has two banks. The instruction-fetch unit can only fetch words from the first bank, and the execution unit can read from and write to both banks. We adopted this scheme so that the execution unit could execute load and store instructions that access the second bank completely in parallel with the instruction-fetch unit reading words from the first bank. The first bank uses an arbiter to allow shared access between the instruction-fetch unit and the execution unit.

5 Preliminary Results

5.1 Network-Simulation Results

Before designing SNAP, we designed asynchronous circuits that simulate two networks using different medium access control (MAC) protocols. The first network contains nodes that use the slotted Aloha MAC protocol and multiple-packet reception radios [23], while the second contains nodes that use the IEEE 802.11 distributed coordination function (DCF) [8] MAC protocol. Designing and simulating these circuits validated our idea that we could use QDI asynchronous circuits to simulate sensor networks [16].

The first circuits we designed simulated the slotted-Aloha networks described in [23]. The values of certain statistics (end-to-end throughput, latency, etc.) that we obtained from simulatin these circuits exactly matched those produced by the Matlab program used for [23]. The circuits we created for these simulations were very similar to the NoC: They consisted of grids of processing elements (each of which simulated a single node in the network) that communicated by passing messages via a mesh interconnect. The main difference is that the processing elements were custom circuits (which implemented the slotted Aloha protocol) instead of general-purpose SNAPs.

The circuits that simulated the 802.11 network were similar to those that simulated the Aloha network, except that the processing elements implemented the 802.11 MAC protocol. Our results matched exactly against those produced by the simulator used by the authors of [1].

Results from [10] indicate that the execution time of a TBS-based sensor-network simulator running on the NoC should remain constant as we vary the number of nodes: We estimate that the NoC will be able to simulate typical network scenarios 22 times faster than real time, regardless of the number of simulated nodes. Such scalability would be much better than that of conventional parallel simulators [36]. Part of our future work is to evaluate the performance of more traditional network simulators running on the NoC.

The factor that ultimately limits the NoC's time scale is the ratio of the physical time required to pass messages between SNAPs via the NoC's interconnect to the time for a simulated node to change its antenna from receiving mode to transmitting mode (the latter time is specified as $5\mu s$ for nodes using the IEEE 802.11 DCF [8]). Decreasing the NoC interconnect's latency or increasing the simulated nodes' transmitter-turn-on time will increase the time scale of the simulation [10].

5.2 Processor-Performance Results

An early, unoptimized version of our instruction-fetch unit operates with a cycle time of between 30 and 40 transitions, depending on the value of the program counter (the latency of the incrementer depends on the value of the program counter). Preliminary SPICE results from an auto-generated layout with a single fixed stack size for each gate suggest a throughput of 326MHz in TSMC's $0.18\mu m$ process for this fetch unit. Optimizing the layout should allow us to decrease SNAP's cycle time greatly. The DRAM memory should not be a limiting factor as we have demonstrated an interleaved on-chip memory running at over 500MHz [5]. In addition, our simulations show that the SNAP nodes will be idle 80-90% of the time across a variety of network scenarios. Therefore, SNAP will spend most of its time waiting to complete a `DONE` instruction. The asynchronous nature of the processor will cause all switching activity to cease in this state, conserving energy for the sensor node. Because we have not yet completed layout of our processor, we do not have any energy or area estimations.

6 Related Work

The work related to this project comes in three flavors: network simulators, multicomputers, and sensor-network nodes.

The most commonly-cited network simulators in papers addressing sensor networks are `ns-2` [22], Glomosim [36], and Opnet [3]. Of these, only Glomosim is a parallel simulator. The only simulators capable of simulating large-scale sensor networks are Glomosim, Qualnet [29], and the Simulator for Wireless Ad Hoc

Networks (SWAN) [12]. Although the performance of these parallel simulators scales with network size much better than the performance of sequential simulators, none of them is able to simulate large-scale networks faster than real time.

The chip multiprocessor which most-closely resembles the NoC is the MIT `Raw` processor [32]; a single `Raw` processor "tile" is analogous to SNAP. However, SNAP adds architectural enhancements specifically designed for network nodes, such as bit-field-manipulation instructions. SNAP also lacks floating point and multiply/divide capabilities, and uses DRAM for its on-chip memory. SNAP bears some similarity to the J-Machine [2] and Mosaic [13], which are message-passing multiprocessors, but which lack the capabilities of the timer coprocessor.

Wireless sensor networks such as those based on Berkeley's Motes [14] utilize low end microcontrollers from Atmel, while more higher performance sensor nodes utilize microprocessors such as Intel's StrongARM [35]. Although using commodity off-the-shelf components for sensor nodes reduces the initial hardware deployment cost, we predict that our use of the same hardware platform for both simulation and deployment will make the end network more predictable and reduce the debugging time. SNAP is also optimized for common operations in network protocols with instructions that allow event queuing.

7 Summary and Future Work

We have presented the design of a Sensor Network Asynchronous Processor (SNAP). This processor will be the basis for a wireless sensor-network node and a component of a chip multiprocessor, the Network on a Chip (NoC), which will be used for sensor-network simulations. We have described how we designed SNAP to meet three main requirements—low area, fast message passing, and efficient event-queue management—that will enable it to serve as both a physical and simulated network node.

At this point we have completed the high-level design and much of the low-level design (production rules [17]) of SNAP. We plan to finish laying out the processor in early 2003. We are also currently designing the physical sensor nodes that will use SNAP, and the simulator that will run on the Network on a Chip.

When we have completed our sensor networks and our network simulator, we plan to use them as a test bed for studying sensor-network applications, and for evaluating the realism of simulated radio and channel layers. We also plan to explore new sensor-network routing protocols that could take advantage of the faster-than-real-time speed of network simulators running on the NoC.

Acknowledgments

This work was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564, and in part by a National Science Foundation CAREER award under contract CCR 9984299.

Thanks to Giuseppe Bianchi, Gokhan Mergen, Jason Liu, and the UCLA Parallel Computing Laboratory for giving us access to the source code for their network simulators.

A SNAP-specific Instructions

Our ISA includes non-standard instructions for manipulating sets of bits within registers, and for managing the event queue. A summary follows.

A.1 Bit-field Instructions

- `BFS dst src1 hi lo.` (Bit-field set)
  `reg[dst]`$[hi : lo]$:= `reg[src1]`.
  The 16-bit immediate indicates the range of `reg[dst]` that should be set to the value contained in `reg[src1]`. This instruction should be useful for constructing messages.

- `BFR dst src1 hi lo.` (Bit-field read)
  `reg[src1]` := `reg[dst]`$[hi : lo]$.
  The 16-bit immediate indicates the range of `reg[dst]` that should be placed in `reg[src1]`. This instruction should be useful for reading messages.

A.2 Event-queue-management Instructions

- `SCHEDULE id, hi, lo.`
  `timestamp_register[id]`$[31 : 16]$:= `reg[hi]`.
  `timestamp_register[id]`$[15 : 0]$:= `reg[lo]`.
  Also turns "on" `timestamp_register[id]`.

- `CANCEL id.`
  Turns "off" `timestamp_register[id]`.

- `TIMESCALE id.`
  Chooses the contiguous 32-bit sample of the incrementer to which the timer coprocessor will compare to all of its timestamp registers. The value of `reg[id]` tells the timer coprocessor which bit of the incrementer will be the lowest-order bit of the sample.

References

[1] G. Bianchi. Performance Analysis of the IEEE 802.11 Distributed Coordination Function. *IEEE Journal on Selected Areas in Communications*, 18(3), March 2000.

[2] W. Dally et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, pages 23-39, April 1992.

[3] F. Desbrandes, S. Bertolotti, and L. Dunand. Opnet 2.4: an environment for communication network modeling and simulation. *Proceedings of the European Simulation Symposium*, pages 609–614, Delft, Netherlands, October 1993.

[4] S. Desilva and S. Das. Evaluation of a Wireless Ad Hoc Network, *Proceedings of the 9th Int. Conf. on Computer Communications and Networks (IC3N)*, Las Vegas, October 2000.

[5] V. Ekanayake. Asynchronous Dynamic Random Access Memories. Master's Thesis, Cornell University, Ithaca, NY, 2002.

[6] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM,* 33(10): 30-53, October 1990.

[7] J. Garside et al. AMULET3i - an Asynchronous System-on-Chip *Proceedings Async 2000* pages 162-175 IEEE Computer Society Press April 2000.

[8] IEEE. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Standard 802.11,* June 1999.

[9] G. Kane, J. Heinrich. Mips Risc Architecture. Prentice Hall. 1991.

[10] C. Kelly. Wireless Network Simulation Done Faster Than Real Time. Master's Thesis, Cornell University, Ithaca, NY, 2002.

[11] A. Lines. Pipelined Asynchronous Circuits. *MS Thesis,* California Institute of Technology, 1996.

[12] J. Liu et al. Simulation modeling of large-scale ad-hoc sensor networks. *European Simulation Interoperability Workshop,* 2001.

[13] C. Lutz et al. Design of the Mosaic Element. `http://resolver.library.caltech.edu/caltechCSTR:1983.5093-tr-83`

[14] A. Mainwaring et al. Wireless Sensor Networks for Habitat Monitoring. *ACM International Workshop on Wireless Sensor Networks and Applications,* September 28, 2002.

[15] D. Maltz, J. Broch, and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed. *Technical Report CMU-CS-99-116*, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1999.

[16] R. Manohar and C. Kelly. Network on a Chip: Modeling Wireless Networks with Asynchronous VLSI. *IEEE Communications Magazine*, November 2001.

[17] R. Manohar and A. Martin. Quasi-delay-insensitive circuits are Turing-complete. Invited article, *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems.* March 1996. Available as Caltech technical report CS-TR-95-11, November 1995.

[18] R. Manohar and A. Martin. Pipelined Mutual Exclusion and the Design of an Asynchronous Microprocessor. Cornell Computer Systems Lab Technical Report CSL-TR-2001-1017, November 2001. `http://www.csl.cornell.edu/scripts/listtr`

[19] A. Martin. Synthesis of Asynchronous VLSI Circuits. Caltech Technical Report CS-TR-93-28. 1991.

[20] A. Martin et al. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI,* ed C.L. Seitz, MIT Press, 1989.

[21] A. Martin et al. The Design of an Asynchronous MIPS R3000. *Proceedings of the 17th Conference on Advanced Research in VLSI,* pages 164–181, September 1997.

[22] S. McCanne and S. Floyd. The ns network simulator. Available on the web from the following site: `http://www.isi.edu/nsnam/ns/`.

[23] G. Mergen and L. Tong. Receiver Controlled Medium Access in Multihop Ad Hoc Networks with Multipacket Reception. *Proceedings of 2001 MILCOM*, Washington, DC, Oct. 2001.

[24] N. Paver et al. Register Locking in an Asynchronous Microprocessor. *Proc. ICCD '92*, 351-355, October 1992.

[25] J. Poulton. An embedded DRAM for CMOS ASICs. *Proceedings. Seventeenth Conference on Advanced Research in VLSI.* pp. 288-302, 1997.

[26] Radio Frequency Monolithics (RFM) TR1000 916.50Mhz transceiver chip Datasheet. Available: `www.rfm.com/products/data/tr1000.pdf`

[27] V. Raghunathan et al. Energy aware wireless microsensor networks, *IEEE Signal Processing Magazine,* vol. 19, iss. 2, pp. 40–50, March 2002.

[28] M. Renaudin, P. Vivet and F. Robin. ASPRO-216: A Standard-Cell QDI 16-bit RISC Asynchronous Microprocessr. *Proc. of 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '98).* 1998. pages 22–32.

[29] Scalable Network Technologies. Qualnet. `http://www.scalable-networks.com/`.

[30] M. Takai, J. Martin and R. Bagrodia. Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks.*Proceedings of the 2001 ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2001)*, October 2001. pages 87–94.

[31] Michael Taylor. The Raw Prototype Design Document. `ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf`. 2002.

[32] M. Taylor et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro,* Mar/Apr 2002.

[33] J. Teifel. Interchip Communication in Asynchronous VLSI Systems. *Cornell Computer Systems Lab Technical Report CSL-TR-2002-1027*, October 2002.

[34] Wireless Integrated Network Sensors, University of California, Los Angeles, Available: `http://www.janet.ucla.edu/WINS`

[35] Wireless Sensing Networks Project, Rockwell Scientific. Available: `http://wins.rsc.rockwell.com`

[36] X. Zeng, R. Bagrodia, M. Gerla. GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. *Proceedings of the 12th Workshop on Parallel and Distributed Simulations—PADS '98*, Banff, Alberta, Canada, May 1998.