An Ultra Low-Power Processor for Sensor Networks

Virantha Ekanayake, Clinton Kelly, IV, and Rajit Manohar
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY 14853, U.S.A
{viran, clint, rajit}@csl.cornell.edu

ABSTRACT

We present a novel processor architecture designed specifically for use in low-power wireless sensor-network nodes. Our sensor network asynchronous processor (SNAP/LE) is based on an asynchronous data-driven 16-bit RISC core with an extremely low-power idle state, and a wakeup response latency on the order of tens of nanoseconds. The processor instruction set is optimized for sensor-network applications, with support for event scheduling, pseudo-random number generation, bitfield operations, and radio/sensor interfaces. SNAP/LE has a hardware event queue and event coprocessors, which allow the processor to avoid the overhead of operating system software (such as task schedulers and external interrupt servicing), while still providing a straightforward programming interface to the designer. The processor can meet performance levels required for data monitoring applications while executing instructions with tens of picojoules of energy.

We evaluate the energy consumption of SNAP/LE with several applications representative of the workload found in data-gathering wireless sensor networks. We compare our architecture and software against existing platforms for sensor networks, quantifying both the software and hardware benefits of our approach.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; C.0 [**General**]: Hardware/software interfaces, Instruction set design

General Terms

Design

Keywords

Low-energy, sensor networks, asynchronous, wireless, sensor network processor, event-driven, picojoule computing

ASPLOS'04, October 7–13, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

Our world is becoming increasingly connected and instrumented with sensors. Improvements in microelectronics and integrated systems have made possible sensor platforms ("nodes") that are a few millimeters in dimension [45]. The possible applications of sensor platforms are varied, and include: smart home systems monitoring temperature, humidity, and movement; vibration sensors for earthquake monitoring; stress/strain sensors for monitoring materials and machines; gas sensors for detection of chemical substances; biological sensors for the detection of microorganisms and environmental monitoring; and habitat monitoring to study species in their natural environment.

One of the key issues in the design of these sensor platforms is the power consumption of each component in the node, and in the network as a whole. Nodes typically must be able to perform a combination of computation, wireless communication, and sensing. Each node also contains a power source, which can consist of a conventional battery, a renewable source that generates power using scavenging techniques (e.g. vibration based [17], solar based [5], RF based [6]), a radioactive thin-film that generates high-energy particles [16], or some combination of these ideas, to name some of the possibilities. The lifetime of a sensor network is a function of the operations (computation, communication, sensing) performed by its nodes and of the amount of energy stored in its nodes' batteries.

Conventional wisdom in sensor network design typically focuses on minimizing communication, because conventional communication links consume a significant amount of energy—an amount that contains a term that is dictated by the distance the link must be able to span [10]. However, recent developments in self-powered MEMS-based RF communication devices [13] and in network organization [19] can lead to sensor networks where the communication link is entirely self-powered, shifting the focus to the energy requirements of the computation being performed. The concept of sensor networks with mobile agents treats a collection of sensor nodes as a statistical entity. Instead of thinking of the communication link as something that must be reliable, the shift is to use a statistical treatment that attempts to infer properties of the network based on information from a subset of the sensor nodes and some knowledge about correlations among the monitored data values [19].

This paper presents the design of a low-energy processor—SNAP/LE, for Sensor Network Asynchronous Processor/Low Energy—optimized for data monitoring operations in sensor networks. Instead of minimizing the energy usage of a conventional microprocessor, we have designed a new *asynchronous* microprocessor that contains hardware support for commonly-occurring operations in sensor networks; our goal is to maximize the lifetime of a network. SNAP/LE is event-driven, with extremely low-overhead transitions between active and idle periods. The use of asynchronous circuits results in automatic, fine-grained power management, because circuits that are not required to perform a particular operation do not have any switching activity. Using asynchronous circuits also necessitates that we eliminate glitches or switching hazards in our processor, removing another source of energy waste. The hardware support for event execution in SNAP/LE obviates the need for SNAP/LE to run an operating system. Not having to run an operating system on our processor not only reduces static and dynamic instruction counts, but also allows us to simplify the design of SNAP/LE, which does not need to support functions such as precise exceptions and virtual memory translation. Our final design resembles a microcontroller, and uses an extremely small amount of energy per operation.

This paper builds upon the work in [20], in which we present the design of the SNAP instruction set architecture (ISA). This ISA is suitable for both a chip-multiprocessor network emulation platform as well as a sensor node. We evaluated an implementation of the SNAP ISA that is capable of operating at 326 MIPS, and we showed the expected performance benefits of using such an implementation for emulating wireless networks [20]. This paper evaluates SNAP/LE, an implementation of the SNAP ISA designed to be a stand-alone, low-energy processor for use in sensor network nodes.

Our contributions can be summarized as follows: an event-driven microprocessor architecture that we have implemented with asynchronous circuits, enabling fine-grained automatic power management; a processor capable of fast transitions from active to deep sleep back to active mode; extra hardware that provides an elegant interface between the processor and the sensor node's radio and sensors; a detailed evaluation by instruction type of the energy usage of an asynchronous microprocessor; an evaluation of the energy required to execute common software handlers in sensor networks; and a detailed comparison between SNAP/LE and existing sensor network platforms. To our knowledge, this is the first microprocessor that has been specifically optimized for ultra low-power operation in a sensor network. We expect SNAP/LE to use significantly less energy per instruction than do conventional microcontrollers used by existing sensor network platforms.

We begin by presenting an overview of asynchronous design and by describing some of the ways in which a processor implemented with asynchronous circuits can save energy over a synchronous counterpart (Section 2). We present a description of the architecture of SNAP/LE, including the special hardware support for common operations in sensor networks (Sections 3). We present SPICE-based estimates of the energy per operation and of the execution time of SNAP/LE, and we show how this energy usage is distributed amongst the different parts of the processor. We also estimate the energy required for SNAP/LE to execute typical sensor network applications, and compare these results against those for existing sensor network platforms (Section 4). Finally, we discuss related work (Section 5) and our future work (Section 6).

2. ASYNCHRONOUS DESIGN

Asynchronous systems do not use clocks for sequencing. With the advent of larger and larger dies and faster and faster circuits, the signal propagation delay across a chip has become greater than a single cycle, leading circuit designers to consider asynchronous design for system-level integration approaches. Researchers have proposed globally asynchronous/locally synchronous designs as a way to mitigate global timing issues in clocked systems while still using conventional clocked design in local regions of a chip [14]. Locally asynchronous/globally synchronous designs have also been proposed as a way to use asynchronous designs to improve the performance of local components while integrating them into conventional, clocked system-level architectures [14]. SNAP/LE, on the other hand, is an entirely asynchronous system—there are no clocks in any component of this design.

A clock in a synchronous processor is used for synchronization purposes, and to determine when data is valid (the precise nature of this depends on the clocking discipline). The use of a clock allows us to think of the computation as being a sequence of operations, each taking the same amount of time. An asynchronous circuit must use additional circuitry to synchronize communication. The clock signal is also used to determine when a particular signal should be examined. The absence of a global clock implies that a circuit observing a particular signal cannot ignore spurious transitions on that signal (glitches or switching hazards). Therefore, every signal in an asynchronous design must be hazard-free. The additional circuits required for synchronization constitute the overhead incurred by adopting an asynchronous approach.

Asynchronous circuits use a *handshaking protocol* to implement synchronization. Data validity can be encoded in a variety of ways, depending on the particular nature of the asynchronous circuit. We have designed SNAP/LE using *quasi delay-insensitive* (QDI) circuits. The QDI design style is arguably the most conservative asynchronous design style [9]; a QDI asynchronous circuit is guaranteed to operate correctly regardless of the delays in the devices used to implement the circuit, as long as the circuit satisfies a minor relative timing assumption on the delay of some wires that connect the output of a gate to the inputs of multiple gates (known as an *isochronic fork* assumption) [3]. This assumption is necessary to build non-trivial asynchronous circuits, and its incorporation permits the design of arbitrary asynchronous computations [4].

Although asynchronous QDI designs do not include any clocks, there is a well-defined notion of *cycle time*, which is the inverse of the rate at which instructions are executed by the processor [7, 8]. The difference between the clock cycle time and the asynchronous cycle time is that the latter can vary depending on the dynamic state of the processor and the operation being performed.

3. SNAP ARCHITECTURE AND ISA

Modern sensor network nodes typically use commodity off-the-shelf (COTS) microcontrollers. The Berkeley Motes, for example, use the Atmel Mega128L [26]. Although such microcontrollers provide the functionality necessary to serve as the processing elements in sensor networks, a processor designed *specifically* for sensor networks will be able to meet the computational demands of a sensor network node while

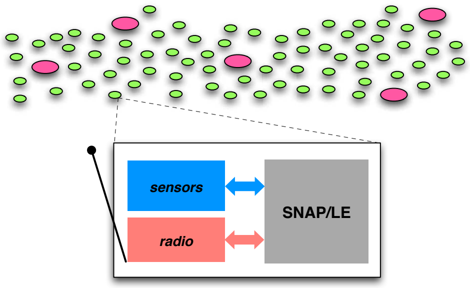

Figure 1: A sensor-network node that uses SNAP/LE. Larger nodes have more resources (e.g., aggregation points).

consuming much less energy.

Before designing a custom processor for sensor networks, we must define what characteristics we want such a processor to have. We identified the following properties as desirable for such a processor:

A lower-power sleep mode. The data-gathering sensor nodes for which we designed SNAP/LE will be idle much of the time. Therefore, the power consumption of SNAP/LE while it is "asleep" (not computing) will play a large role in determining the battery life of the nodes.

A low-overhead wakeup mechanism. When an event, such as the arrival of a radio packet, occurs, SNAP/LE will wake up, execute some code, and then go back to sleep. Ideally, the time spent waking up and going to sleep should be much less than the time spent executing code. We estimate that SNAP/LE will require on the order of microseconds to execute the code to handle a given event (Section 4); we would therefore like SNAP/LE to be able to wake up or go to sleep in tens of nanoseconds.

Low power consumption while awake. Although the power consumption of the processor while it is "asleep" will play an important role in determining the battery life of a node, we also want the processor to be energy-efficient while it is "awake" (computing).

A simple programming model. Finally, we want programmers to be able to use SNAP/LE easily. The nodes in sensor networks described in Section 1 will be asleep most of the time, periodically waking up to handle radio traffic or sensor data. The programming model should map easily to this behavior. Moreover, the programmer (or compiler) should be able to easily implement actions that a sensor network node frequently performs, such as scheduling internal timers or reading sensor data.

We designed SNAP/LE with these features in mind. The essential difference between SNAP/LE and a conventional microprocessor or microcontroller is that SNAP/LE is *event-driven.* The processor begins by executing some boot code, and then waits for an *event* to occur, at which point it executes the appropriate *event handler.* We can therefore consider the processor to have two states: *awake* (executing the boot code or an event handler) and *asleep* (waiting for an event to occur).

Figure 1 shows the organization of a sensor network node that uses SNAP/LE. The *external events* to which SNAP/LE can respond are incoming radio traffic and sensor readings. The processor may also schedule timeouts, for example to wake itself up periodically to take sensor readings. The end of such a timeout is an *internal event.*

Conventional embedded microprocessors often use software to implement event-driven behavior. For instance, the well-known Berkeley Mote platform uses the TinyOS software layer to provide this functionality. TinyOS is not an operating system in the traditional sense; rather, it provides a set of software components that abstracts a hardware interrupt as an event, and implements a simple FIFO task scheduler. Although this approach allows the use of commodity processors in event-driven sensor nodes, it introduces an extra software overhead that is not present in a processor such as SNAP/LE, which we designed for event-driven operation.

Figure 2 shows the architecture of SNAP/LE. SNAP/LE consists of three components: (1) The timer coprocessor, which schedules internal timeouts , (2) the message coprocessor, which provides an interface to the sensor node's sensors and radio, and (3) the processor core, which consists of the event queue, instruction fetch, decode, execution units, busses, register file, message FIFOs, and memories (everything in Figure 2 other than the two coprocessors, in other words).

3.1 The Processor Core

SNAP/LE has an in-order, single-issue core that does not perform any speculation. It has a 16-bit datapath; instructions can consist of one or two 16-bit words (two-word instructions take two cycles). Although SNAP/LE is not highly-pipelined at the circuit level, it can support the simultaneous execution of several instructions. Figure 2 shows the potential concurrency of the processor; the solid circles correspond to instruction tokens in-flight during execution (the circles in the event queue correspond to outstanding events that are yet to be processed). These instruction tokens travel through the pipeline and are transformed by the computation blocks (adders, decoders, etc.) through which they move. Because we have designed SNAP/LE with asynchronous circuits, each computation block will have no switching activity until a token arrives at one of its inputs. This data-driven switching activity reduces the total switching capacity of the processor, saving energy. These energy savings come as a direct result of our use of QDI asynchronous circuits; to achieve equivalent savings in a clocked processor, the designer would have to clock gate every latch in the processor.

The most interesting components of SNAP/LE's processor core are the event queue and the instruction fetch, which combine to form a hardware implementation of a FIFO task scheduler. SNAP/LE begins by executing its boot code. The last instruction in the boot code is the special "`done`" instruction, which tells the instruction fetch to stop fetching instructions and to wait for an *event token* to appear at the head of the event queue. Event tokens can be inserted into the event queue by the timer coprocessor (when a timeout finishes) or by the message coprocessor (when data arrives from the sensor node's radio or one of its sensors). Each event token contains information that indicates which event occurred.

If an event token is present at the head of the event queue, the instruction fetch will remove it and use it as in index into SNAP/LE's *event-handler table* to determine the address of the appropriate event handler. SNAP/LE will execute instructions starting at that address until it reaches another `done` instruction, at which time it will check the event queue again.

If the event queue is empty when SNAP/LE executes a `done` instruction, the fetch will stall until an event token appears. As soon as the core finishes executing the instructions that proceeded the `done` instruction, all switching activity in the core will stop, and the processor will be "asleep." Thus we have met one of our design goals: to have a low-power sleep mode.

The time required for the processor to wake up and begin executing a new event handler is merely the time for an event token to propagate through the event queue. This time is on the order of tens of nanoseconds. Therefore, we have met another of our design goals: The time for the processor to wake up and fall back asleep is much less than the time that the processor will spend executing event handlers.

Conventional processors and microcontrollers typically have several sleep states. A processor in a "deeper" sleep state uses less power but requires more time to wake up than a processor in a "lighter" sleep state. The Atmel microcontrollers, for example, have six sleep states [26]. SNAP/LE, on the other hand, only has a single sleep state, during which all switching activity in the core stops. The time to wake up from this sleep state is on the order of tens of nanoseconds. Thus we get the best of both worlds: the energy savings of "deep" sleep with the low wake-up latency of "light" sleep.

The SNAP ISA is designed to execute handlers, and not a full operating system. Therefore, SNAP/LE does not support precise exceptions, which have additional overhead even in in-order issue asynchronous processors due to the information necessary to reconstruct program order after the decode issues instructions to different execution units [39]. The SNAP ISA does not support virtual memory or even interrupts. All external events are processed through the event queue, which prevents an event handler from being preempted by the occurrence of a new event. This guarantees atomicity of handler execution, eliminating any concurrency management issues that could arise due to handler interleaving. Interrupts that would normally be required for off-chip interfaces are translated into events by the timer and message coprocessors, thereby removing the need to support

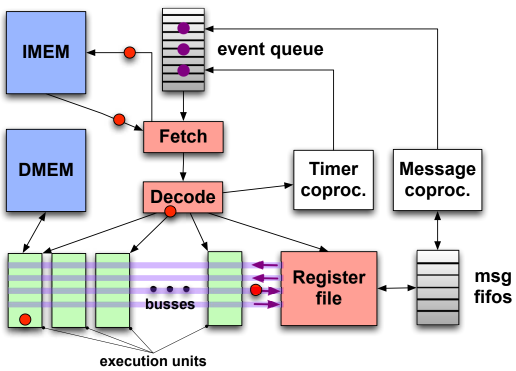

Figure 2: Microarchitecture of SNAP/LE showing major units. The circles correspond to data tokens that are in-flight instructions or pending events (in the event queue).

exceptions or interrupts in the processor core.

SNAP/LE has two on-chip 4KB memory banks and no caches. The first bank, the IMEM, contains instructions; the other, the DMEM, contains data. The core can write to either the IMEM or the DMEM, allowing it to modify its own code (and also providing a way to bootstrap the processor by sending it code over the radio link). In an embedded environment, SNAP could be designed with its IMEM implemented partially or entirely using ROM, but we are designing it using RAM so that we can run many different experiments on the chip once it is fabricated.

One advantage of building SNAP/LE with QDI asynchronous circuits is that we can optimize its design for average-case, rather than worst-case, behavior [40]. As an example, consider the busses in SNAP/LE. Instead of attaching all of the execution units directly to one set of busses, we have a two-level bus hierarchy that consists of one set of "fast" busses and one set of "slow" busses. The most commonly-used execution units connect directly to the fast busses, and those that are used less often connect to the slow busses. The fast busses communicate directly with the register file, whereas the slow busses communicate with the register file through the fast busses. This scheme improves the datapath's average-case performance by dramatically decreasing the amount of capacitance on the fast busses [40]. Moreover, because SNAP/LE uses asynchronous circuits, we do not have to add any additional logic to handle the timing variations introduced by the different busses (or by the varying, data-dependent throughput of different execution units). The Lutonium asynchronous processor uses a similar hierarchical bus scheme [2].

The processor core has the following execution units: an adder, a logic unit, load-store units for the IMEM and the DMEM, a timer unit for interfacing with the timer coprocessor, a jump/branch unit, a linear-feedback shift register (for pseudo-random number generation), and a shifter. Our studies of sensor network code suggest that the most commonly-used units will be the adder, the logic unit, the load-store unit for the DMEM, the shifter, and the jump/branch unit, so we placed those on the fast busses and the rest on the slow busses. We designed all of the functional units with little internal pipelining. Such a design style somewhat limits the throughput of our processor, but it also helps fulfill our goal of limiting SNAP/LE's power consumption while awake.

3.2 The Timer Coprocessor

SNAP/LE uses the *timer coprocessor* to schedule timeouts. The timer coprocessor consists of three self-decrementing, 24-bit *timer registers.* To schedule a timeout, the processor sends to the timer coprocessor both a timer number (indicating which of the three timer registers to use) and a timeout duration. The timer coprocessor then sets the appropriate timer register and the register begins decrementing itself. If necessary, the decrementing frequency can be calibrated against a precise timing reference.

When a timer register decrements itself to zero ("expires"), the timer coprocessor inserts an event token into the event queue (the token contains information indicating which of the three timer registers expired, since each has its own entry in the event-handler table). Timer registers that are not decrementing have no switching activity, and therefore use little energy.

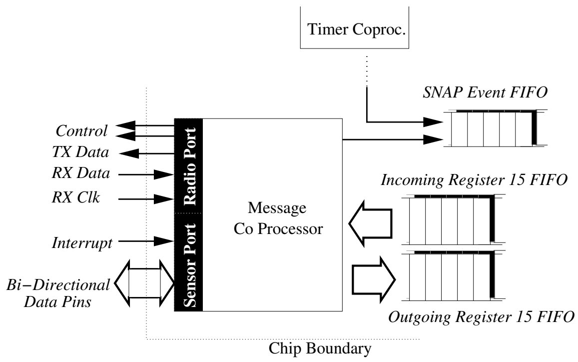

Figure 3: **The Message Coprocessor**

The core can also cancel a previously-scheduled timer register. To avoid the race condition in which the core attempts to cancel a timer register that has already expired, we have designed the timer coprocessor such that it inserts tokens into the event queue for canceled timer registers as well as for those that have expired. The software running on the core must therefore maintain information about which timer registers it has canceled.

3.3 The Message Coprocessor

The message coprocessor, shown in Fig. 3, is the interface between the processor core and the external sensors and radio. All communication with the radio and sensors is handled through two 16-bit message FIFOs (shown in Figure 3) that map to general-purpose register 15 (`r15`), an I/O scheme similar to that used in the Mosaic Element [11] and in the RAW microprocessor [12], among others. SNAP/LE's register file actually has only fifteen physical registers. An instruction that writes a value to `r15` will cause the processor core to insert a value into the message coprocessor's incoming FIFO, and an instruction that reads from `r15` causes the processor core to remove an entry from the head of the message coprocessor's outgoing FIFO. When an external event—such as the reception of a word from the radio or a new sensor reading—occurs, the message coprocessor inserts a token into SNAP/LE's event queue.

Communicating with a standard radio. For our first prototype SNAP/LE sensor node we plan to use a commercial radio transceiver, which will be a separate off-chip component (in our case, the RFM TR1000 used in Berkeley Motes [22]) interfaced through two control pins that select the radio mode, a serial transmit data output, a serial receive data input, and a radio clock input. This interface is general enough to interface with other radio transceivers as well [23].

To receive radio transmissions, a program writes an *RX* command word to `r15`, signaling the message coprocessor to configure the radio for reception. Once the radio detects an incoming transmission and sends a 16-bit word to the processor, the message coprocessor will notify the core via the event queue, and insert the data into its outgoing `r15` FIFO. We chose this method of word-by-word message reception because of the relatively slow radio data rates (around 19.2kbps), which would otherwise stall the processor core for almost a millisecond per word. It is, however, a much more efficient scheme than the bit-by-bit interrupt scheme most microcontrollers use to receive data [26], because the bit/word conversions are handled by the message coprocessor, leaving the core free to process other events. The scheme for transmission is similar to that for reception, except that the core writes a *TX* command followed by a data word to `r15`.

Communicating with sensors. Communicating with the sensors can be active (driven by polling) or passive (driven by external interrupts). If a sensor asserts the external-interrupt pin connected to the message coprocessor, the message coprocessor will insert a token into SNAP's event queue. To poll the sensor data pins, the core sends a *Query* command to the message coprocessor, which will read the sensor data pins, insert the value into the `r15` queue, and send a token to the processor event queue.

3.4 Instruction Set Summary

The architecture described above requires several extensions to conventional instruction sets. The SNAP ISA contains instructions that can be broadly classified into five categories: (1) standard instructions—arithmetic, logic, jump, etc.—that exist in most RISC ISAs; (2) instructions for communicating with the timer coprocessor; (3) instructions for communicating with the message coprocessor; (4) instructions that perform operations used frequently by network-protocol code; and (5) utility instructions necessary for the event-driven nature of SNAP.

Standard RISC Instructions. The SNAP ISA includes typical RISC instructions, including arithmetic and logic instructions, jump and conditional branch instructions, and memory-access instructions (each bank has its own load and store instructions). Because our datapath is only 16-bits wide, our ISA includes add-with-carry and subtract-with-carry instructions to enable arithmetic operations with operands containing more than sixteen bits.

Timer Coprocessor Instructions. Because the timer registers are 24-bit and the datapath is 16-bit, we provide two instructions, `schedhi` and `schedlo`, to schedule timers. These instructions set the highest-order 8-bit and lowest-order 16-bit sections of a particular timer register, respectively. The format for the `schedhi` instructions is: "`schedhi $tsreg, $val`." The value in `$tsreg` specifies the timer register number, and the value in `$val` specifies the highest-order eight bits of the value at which the timer register will begin (the format for the `schedlo` instruction is similar). The programmer can cancel a timer with the `cancel` instruction: "`cancel $tsreg.`"

Message Coprocessor Instructions. The SNAP ISA does not include special instructions to send/receive data to/from the message coprocessor. Instead, *any* instruction can communicate with the message by using `r15`. As discussed in Section 3.3, whenever an instruction uses `r15` as a source operand, the instruction reads a 16-bit word from the message coprocessor. Likewise, any instruction that uses `r15` as a destination operand writes its result to the message coprocessor.

Network-Protocol Instructions. The SNAP ISA contains three instructions that perform actions that are commonly used in network code. The first is the bit-field set (`bfs`) instruction, which sets a specific field of a register to a given value. The format of the instruction is: "`bfs $dst`

`$src mask`." The `mask` is a 16-bit value (`bfs` is a double-word instruction) that indicates which bits of `$dst` should be set to the corresponding bits in `$src`. We added this instruction because the software that controls radio communication often needs to modify small fields within words.

Our second instruction, `rand`, generates a pseudo-random number and places the value into the specified register. Finally, we also have the ability to seed the random number generator via the `seed` instruction.

Event-driven Execution Instructions. Our ISA also has two instructions that control the event-driven execution of the processor. The programmer uses the `done` instruction to indicate the end of a handler. As described earlier, when the decode receives a `done` instruction, it sends feedback to the instruction fetch telling it to look for another token at the head of the event queue. (The `done` instruction has no operands.) The `setaddr` instruction changes the values in the event handler table.

4. RESULTS

Evaluation of a clocked design consists of two tasks: determining the number of cycles for executing a task, and determining the clock frequency. These two numbers are coupled in an asynchronous design. To provide an accurate evaluation of SNAP/LE, we used a very detailed low-level simulation of the processor core. Our comparisons are primarily made against the Atmel microcontrollers used by existing sensor network platforms, like the Berkeley motes. While we plan to use SNAP/LE at 0.6V, we provide an evaluation for 0.6V, 0.9V, and 1.8V operation.

4.1 Energy Estimation Methodology

We have currently designed the SNAP/LE processor core at the transistor level. The processor contains approximately 57K transistors, plus an additional 325K for the memories. We used our transistor-level implementation to derive energy estimates from both SPICE and a switch-level circuit simulator. First, we performed a detailed SPICE simulation of a component of the processor core using extracted, automatically generated layout in TSMC's 180nm CMOS process available through the MOSIS VLSI service. We did not perform precise transistor sizing; instead, we used small transistor sizes with fixed nfet and pfet widths for all the logic (non-memory) transistors. Note that accurate sizing that is currently underway will improve the delay for a fixed energy budget, and vice versa. The results from SPICE simulation were used to back-annotate the full, switch-level simulation, and tuned to match SPICE-reported energy estimates from the layout to within 10%. Once we had calibrated the switch-level simulator against SPICE, each application was run to completion using the switch-level simulator and then the total energy and delay reported was divided by the total dynamic instruction count to obtain estimates of the energy per instruction and the processor throughput. This process was repeated for three different operating voltages: 1.8V (the nominal voltage in TSMC's 180nm process), 0.9V, and 0.6V.

4.2 Benchmarks

To gain some insight into the typical software that sensor nodes execute, we created a suite of benchmark programs that corresponded to typical operations and protocols used to evaluate sensor networks by the networking community. To simplify the application development process, we ported `lcc` [34], a freely available retargettable C compiler, to the SNAP ISA, and also developed a complete custom assembler/linker tool-chain. Note that we did not introduce any optimizations in `lcc`, and we present results from unoptimized code that was generated by `lcc`.

For library support, we wrote an IEEE 802.11-based [33] MAC scheme and a simplified routing layer based on AODV, a commonly-used ad-hoc routing protocol [32]. These protocols are typically used by the sensor network community for evaluating new designs. In addition, because SNAP/LE nodes are meant to function as data gathering nodes rather than heavy-duty processing nodes, they will spend most of their time transmitting and relaying radio messages. Therefore, the MAC and routing layers are probably the most critical components that a node will run. Thus, we would like to explore the following: (i) How large (in terms of bytes) are the handlers for receiving, transmitting and routing messages? SNAP/LE has a limited instruction memory and should not have to implement a complex message protocol. (ii) What is the average dynamic instruction count of a handler, and what types of instructions are most frequently used? If a handler takes too long to execute, SNAP/LE may end up dropping pending events because the event queue has filled up. (iii) How much energy does the typical handler consume? Although a complete discussion of the MAC and AODV protocol are beyond the scope of this paper, we present a brief explanation of the handlers for which we gathered statistics to answer the above questions.

Packet Transmission. The MAC layer implementation takes a message from the application layer, and transmits it byte-by-byte across the radio interface.

Packet Reception. The MAC layer implementation handles radio word arrival events, and assembles the complete message.

AODV Route Reply. The MAC receives (as detailed in Packet Reception above) and passes up to the routing layer a route-lookup request from a neighboring node. A lookup is then performed in the node's routing table and a route reply (RREP) message is sent to the MAC layer, where packet transmission occurs.

AODV Packet Forward. The MAC receives a data packet destined for another node. The routing layer looks up the next hop in its routing tables and sends this updated packet for transmission by the MAC.

In addition to the above handlers that constitute core functionality that would be implemented in such a sensor node, we also wrote two simple sensor applications.

Temperature Sense. Simulates reading a sensor and computing a running average and logging the value.

Range Comparison. Simulates receiving a packet, comparing two fields, and logging the larger of the two.

These short applications are typical of the types of operations done in real life sensor networks such as habitat monitoring experiments [29].

In addition to analyzing the performance of SNAP/LE when executing these programs, we provide three comparisons against the MICA motes running the TinyOS platform. We obtained instruction-count statistics for TinyOS

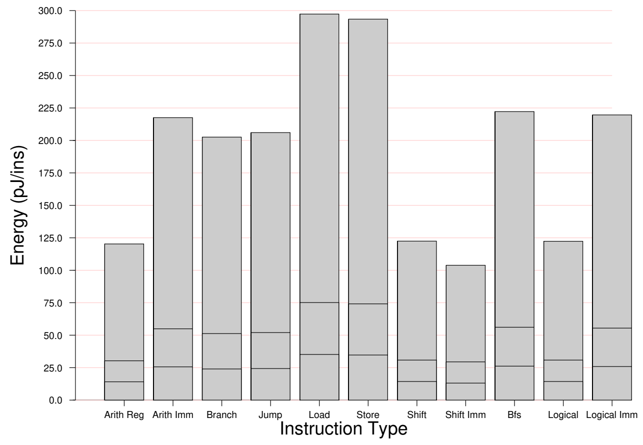

Figure 4: Energy per instruction type

applications by using execution-driven simulation on a cycle-accurate ATMega 128 simulator (AVR Studio, available from Atmel). For a compiler, we used the AVR `gcc` tool-suite, which performs many more optimizations than the `lcc` compiler we use for SNAP/LE.

We provide an analysis of two example applications from TinyOS: (1) BlinkTask [41], which sets up a periodic timer interrupt that enqueues a function on the TinyOS task queue to blink an LED. (2) SenseTask [41], that periodically samples a data input, computes a running average, and displays the result. We also provide a comparison against the MICA high-speed communications stack [42], which we reimplemented for the SNAP ISA.

4.3 Performance and Wake-up Latency

The average throughput of SNAP/LE was measured using the benchmark programs above at three different operating voltages. At 1.8V, SNAP/LE can sustain an average instruction execution rate of 240 MIPS. This rate drops by a factor of four at 0.9V, to 61 MIPS, and by a factor of 8.6 at 0.6V, to 28 MIPS. For comparison, the Atmel Mega128L operates at 4MIPS with a 3V nominal power supply [26].

SNAP/LE's transition from idle to active was measured to be eighteen gate delays, which corresponds to a 2.5ns wake-up latency at 1.8V, a 9.8ns wake-up latency at 0.9V, and a 21.4ns wake-up latency at 0.6V. Contrast this to the Atmel microcontrollers, which require between four and sixty-five milliseconds to transition from sleeping to active [26].

4.4 SNAP/LE core energy consumption

In Fig. 4, we show the energy consumption per instruction class for the more commonly executed instructions. We derived these estimates by running programs of one thousand of each instruction using uniformly distributed random operands, and averaging across the type of instruction. The energy estimates include the instruction and data memory energy consumption, based on an asynchronous on-chip memory [18]. Each bar in the figure has three horizontal marks, from top to bottom, corresponding to running SNAP/LE at 1.8V, 0.9V, and 0.6V respectively.

Running at its highest voltage, the SNAP/LE core consumes under 300pJ per instruction. Contrast this energy figure against the Atmel microcontroller, which uses approximately 1500pJ/ins. When running with a 0.6V supply voltage, SNAP/LE uses less than 75pJ/ins, with many instruction types using less than 25pJ/ins.

There are three distinct instruction types in terms of energy per instruction. Instructions like “Arith Reg” and “Shift” are represented with sixteen bits (one word), and do not access data memory. These use the least energy per operation, as can be expected. The next category includes instructions that have an additional 16-bit immediate (two-word instructions) such as “Arith Imm” and “Logical Imm.” The most inefficient instructions are the memory operations that require both two instruction memory words as well as a data memory access.

An analysis of the energy distribution in the processor core (excluding instruction and data memories) reveals that 33% of the energy is used by the processor datapath (including the data busses), 20% by the instruction fetch, 16% by the instruction decode, 9% by the memory interface, and 22% by various miscellaneous logic including decoupling buffers and some control circuits. On the whole, this constitutes about half the energy per instruction; the other half is required for memory access.

4.5 Energy Consumption: Benchmarks

Table 1 shows our benchmarks, their total dynamic instruction counts (including all the libraries/etc required for execution on SNAP/LE), the total energy required for each benchmark to run to completion at three different voltages, and the average energy per instruction for each benchmark. The number of instructions executed by the handlers is exceedingly small, allowing the processor to handle the next event after only a short duration. The energy consumption for an entire handler at even the highest voltage is only in the tens of nanojoules. The total code size for the application examples in Table 1 is 2.8KB, which leaves room for additional program code.

The most frequently executed instructions are of type “Arith Reg,” which is the one-word instruction and therefore in the lowest energy-utilization category. The next most frequently used instruction was “Load.” An analysis of the output of `lcc` showed that the reason for this was poor optimization; the compiler generated a lot of load/store operations that were unnecessary (saving/restoring registers). Optimizing the output of `lcc` is a subject of our current investigations.

4.6 TinyOS Benchmarks

To demonstrate the efficiency of our hardware event scheduling support, we present comparisons against three sample applications on our system versus the widely available Berkeley MICA motes running TinyOS. First, we studied a simple example application from TinyOS called BlinkTask [41], which sets up a periodic timer interrupt that enqueues a function on the TinyOS task queue to blink an LED. We used the AVR simulator from Atmel to examine the instruction statistics for the main interrupt and scheduling loop. We also wrote a version of an identical “Blink” application for the SNAP ISA using C (184 bytes of code for the SNAP ISA, versus 1.4KB for the TinyOS version). Following the same flow as TinyOS, we schedule a periodic timer event that enqueues a handler to blink the LED. In SNAP/LE, this operation corresponds to a write to the sensor port.

The results of the analysis are shown in Fig. 5. On the

Table 1: Handler code statistics with energy numbers running on SNAP/LE

Software Task	Dynamic Insts.	1.8V		0.9V		0.6V	
		E (nJ)	E/Ins(pJ)	E(nJ)	E/Ins(pJ)	E(nJ)	E/Ins(pJ)
Packet Transmission	70	15.1	216	3.8	54	1.6	24
Packet Reception	103	22.5	218	5.6	56	2.5	24
AODV Route Reply	224	48.1	215	12.0	54	5.2	23
AODV Forward	245	53.7	219	13.5	55	5.9	24
Temperature App	140	30.5	218	7.7	55	3.4	24
Threshold App	155	33.7	217	8.5	54.7	3.8	24

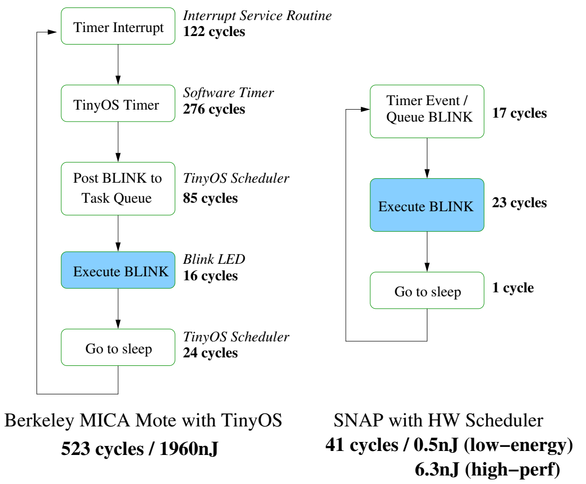

Figure 5: Comparison of a periodic LED Blink program, showing the scheduling overhead of TinyOS

mote, only 16 active processor cycles actually perform the blinking—the other 507 cycles are taken up in handling timer interrupts and the TinyOS scheduler. In contrast, the version on the SNAP ISA, which utilizes the timer coprocessor and the event queue, takes only 41 cycles to complete. When combined with the dramatically better energy per instruction for the SNAP/LE, our implementation uses 0.5nJ at 0.6V or 6.8nJ at 1.8V per blink operation, whereas the TinyOS/Atmel microcontroller combination requires 1960nJ for the same operation.

For our next test, we ported a data-logging and averaging application from TinyOS (called "Sense"), which periodically samples a data value from the analog-digital-converter (ADC), computes a running average, and displays the high order bits on the LEDs. The mote version executes 1118 cycles for a single sample, average, and display iteration, with the interrupt service and scheduler overhead consuming 781 cycles (over 70% overhead), due to the multiple interrupts (timer, ADC). In contrast, the SNAP version needs to execute only 261 cycles in total, which demonstrates how much extra code TinyOS needs to create an event-driven programming interface for a standard microcontroller.

Finally, we ported the TinyOS MICA RFM TR1000 radio-based high-speed communications stack [42] to the SNAP ISA. This stack takes the main TinyOS networking message format and provides SEC-DED error coding and packet CRC, as well as a byte-level interface to the radio via SPI serial signaling. The original code is moderately complex and computationally intensive, comprising about a dozen TinyOS code components; porting this code demonstrates how the SNAP ISA can efficiently implement radio messaging via its event support and radio coprocessor. To send a single data byte via this protocol (including the processing necessary to error encode and calculate CRCs), the TinyOS version requires approximately 780 Atmel cycles. The SNAP/LE version consumes 331 cycles, a 60% reduction despite the unoptimized code emitted by our compiler. When studying the software overhead in TinyOS, we found that the TinyOS interrupt service routine consumes approximately 30% of the cycles; this overhead becomes quite significant when dealing with long messages. Additionally, the need to handle the SPI interface on a byte-by-byte basis on the mote—as opposed to using the two-byte wide radio coprocessor hardware interface on SNAP/LE—also requires additional code in the Atmel implementation.

4.7 Results Summary

SNAP/LE can execute typical handlers suitable for data-monitoring operations using approximately 218pJ/ins at 1.8V and as little as 23pJ/ins at 0.6V. Combining this with dynamic instruction counts in the range of 70-250, SNAP/LE can execute a handler with between 15nJ and 55nJ at 1.8V, and between 1.6nJ and 5.8nJ at 0.6V. For extremely low activity levels (less than ten events per second), this corresponds to active power in the realm of 150nW to 550nW at 1.8V, and 16nW to 58nW at 0.6V—many orders of magnitude less than what we expect from using a conventional microcontroller like the Atmel ATMega128L. This improvement in energy comes in conjunction with extremely low wake-up latencies: on the order of nanoseconds instead of milliseconds.

SNAP/LE also compares favorably from a systems programming point of view, providing the ease of straightforward C-language applications, while removing the overhead of software interrupt service routines and schedulers. Compared with TinyOS, applications on the SNAP/LE processor take fewer cycles to execute due to the lack of scheduling overhead, and even processor-intensive tasks such as byte-level error-encoding can be more efficiently implemented.

5. RELATED WORK

The work presented here builds on the work in [20], which presented a parallel network simulator that runs on a custom chip multiprocessor. The processors in the chip multiprocessor use the SNAP ISA and operate at 326 MIPS.

Table 2: Related Microcontrollers

Processor	Clocked	Speed	Datapath (bits)	Memory	Voltage	E/ins (pJ)
Atmel Mega128L[26] AVR RISC core used by MICA2 Mote, MEDUSA-II	Yes	4MIPS	8	4-8K	3V	1500
Intel XScale[25] High end ARM cores, used in Rockwell sensors, Intel Mote [25, 27]	Yes	200-400MIPS	32	16-32MB	1.3-1.65V	890-1028
Dynamic Voltage Scaled Microprocessor [36] Custom ARM8	Yes	7-84MIPS	32	16KB	1.8-3.8V	540-5600
CoolRISC [24] XE88 microcontroller	Yes	1 MIPS	8	22KB	2.4V	720
Lutonium [2] 8051 compatible in TSMC 0.18μm	No	200MIPS	8	8KB	1.8V	500
Aspro-216 [28] Custom async. microcontroller in STM 0.25μm	No	25-140MIPS	16	64KB	1.0-2.5V	1000-3000
SNAP/LE - 0.18μm TSMC process	No	28MIPS	16	8KB	0.6V	$\approx$24
SNAP/LE - 0.18μm TSMC process	No	240MIPS	16	8KB	1.8V	$\approx$218

The J-machine used a message-driven processor to optimize the execution of remote methods [15]. The dispatch was based on the program counter contained in the message. MAGIC, the programmable coherence-protocol controller in the Stanford FLASH multiprocessor [21]., contained a table-driven dispatch scheme using handlers, which is similar to SNAP/LE's event-handler table scheme SNAP/LE, however, is a single-issue asynchronous core optimized for energy-efficient operation, whereas MAGIC contains a statically-scheduled, dual-issue synchronous core with data buffers that supports efficient execution of coherence protocols. Most low-energy synchronous embedded processors shown in Table 2 require between 500pJ and 1500pJ per instruction across a variety of performance points. Other asynchronous QDI microprocessors include the first asynchronous microprocessor [38], the MiniMIPS [1], the Lutonium [2], and the ASPRO-216 [28]. Asynchronous processors that use other circuit-design styles include the family of Amulet processors that correspond to asynchronous ARM cores [37]. None of these asynchronous processors have the event, radio, or sensor support provided by the SNAP ISA, and the Amulet and MiniMIPS processors require on the order of 1nJ per instruction. Table 2 provides a comparison against two low power asynchronous microcontrollers that are implemented with QDI asynchronous circuits. Apart from the obvious ISA differences between SNAP/LE and the other asynchronous processors, SNAP/LE uses less pipelining than either the Lutonium or the Aspro-216.

There are numerous sensor network testbeds currently in use by networking researchers. Most tend to use COTS microcontrollers. The Berkeley Mote [35] platform uses low-cost Atmel 8-bit AVR RISC microcontrollers such as the ATMega128L running between 4MHz and 8MHz. UCLA's MEDUSA-II sensors [30] also use Atmel AVR RISC microcontrollers. These Atmel controllers consume about 1.5nJ/ins, almost 68 times the energy consumption of SNAP/LE at 0.6V (See Table 2), while operating at a much lower performance point. However, they contain on-board peripherals such as ADCs, and, are fabricated on an older technology than our 1.8V 0.18μm process. Higher-end sensor network platforms, such as the Intel Mote [27] and Rockwell sensors [31], use Intel StrongArm/XScale [25] processors. These processors are currently the most energy-efficient high-end commercial microcontrollers available [31]. These ARM-based processors run at 200-400 MIPS and provide a performance comparable to SNAP/LE running at 1.8V, but have a vastly more complicated core and require much more energy per instruction. They have on-chip caches, megabytes of flash-memory, multiple I/O standards, and Bluetooth support; At 1 nJ/ins, they consume about three to five times more energy than SNAP/LE at 1.8V.

After this manuscript was submitted for publication, a custom 8-bit low-energy unpipelined microcontroller from the Smart Dust [44] project was demonstrated that consumed 12pJ/ins running at 1.0V and peak 0.5MIPS (with typical throughputs 0.01-0.1 MIPS). Although the energy consumption is on the same order of magnitude as SNAP/LE, our performance point is about three orders of a magnitude higher.

Of all these platforms, only SNAP/LE contains the custom hardware support to execute handlers efficiently. On-chip message and timer coprocessors offload common operations to custom hardware, and the flexibility of the processor core makes it possible to execute sensor network protocols by simply writing C code that implements the handlers. The combination of asynchronous circuit techniques, hardware assist, and an event-driven programming model makes SNAP/LE an ultra-low-power processor for sensor networks that is fast enough for typical data monitoring sensor network applications.

6. CONCLUSIONS AND FUTURE WORK

This paper presents SNAP/LE, a novel event-driven microprocessor for sensor networks. SNAP/LE can operate off 0.6V power supply consuming about 24pJ/ins at 28 MIPS, executing typical sensor network handlers using only 1.6nJ to 5.8nJ of energy. This dramatic reduction in energy is made possible using hardware/software co-design.

Improving the generated code from `lcc` is a subject of our current investigations. Better transistor sizing would also improve the energy per operation without any loss in performance. We are currently working on getting accurate idle power (leakage) estimates from SPICE, and we hope to have these results soon. Our current processor, at nominal voltage, is typically too fast for the type of sensor network applications we intend to use it for. Even with low-energy transistor sizing and operating at a low supply voltage, the processor is executing in the many millions of instructions per second range, corresponding to many thousands of handlers per second—more than we expect for data monitoring applications which require more in the realm of tens of handlers per second, if not even fewer. We plan to redesign the processor to sacrifice its performance for even lower energy per instruction. Finally, integrating the MEMS-based self-powered RF communication [13] and battery [16] with SNAP/LE is also a direction we are pursuing.

7. ACKNOWLEDGMENTS

The research described in this paper was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564.

8. REFERENCES

[1] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000. *17th Conference on Advanced Research in VLSI*, September 1997.

[2] A.J. Martin et al. The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller *9th IEEE Symposium on Asynchronous Circuits and Systems*, May 2003.

[3] A.J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conf. on Advanced Research in VLSI*, 1990.

[4] R. Manohar, A. J. Martin. Quasi-Delay-Insensitive Circuits are Turing-Complete. *2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems (invited).* March 1996.

[5] B. A. Warneke et al. An Autonomous 16mm^3 Solar-Powered Node for Distributed Wireless Sensor Networks. *Proceedings of Sensors'02.* 2002.

[6] A. Bayrashev, A. Parker, W.P. Robbins, B. Ziaie. Low frequency wireless powering of microsystems using piezoelectric-magnetostrictive laminate composites. *12th International Conference on Transducers, Solid-State Sensors, Actuators and Microsystems.* 2003.

[7] S.M. Burns and A.J. Martin. Performance Analysis and Optimization of Asynchronous Circuits. *Advanced Research in VLSI: Proc. of the 1991 UC Santa Cruz Conference*, 1991.

[8] T.E. Williams. *Self-Timed Rings and their Application to Division.* Ph.D. thesis, Computer Systems Laboratory, Stanford University, May 1991.

[9] J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1994.

[10] T. Rappaport. *Wireless Communications*. Prentice-Hall, 1999.

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

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

[13] H. Li, A. Lal. Radioisotape-Powered Cantilever for Vacuum Sensing with RF Transmission. *Proceedings of 12th International Conference on Transducers, Solid-State Sensors, Actuators and Microsystems.* 2003.

[14] D. M. Chapiro. Globally Asynchronous Locally Synchronous Systems. PhD Thesis, Stanford University, 1984.

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

[16] H. Guo, A. Lal. Nanopower Betavoltaic Microbatteries. *Proceedings of 12th International Conference on Transducers, Solid-State Sensors, Actuators and Microsystems* 2003.

[17] S. Meninger, J. O. Mur-Miranda, R. Amirtharaja, A. Chandrakasan, J. Lang. Vibration-to-electric energy conversion. *Proceedings of the 1999 International Symposium on Low power electronics and design.* 1999.

[18] V. N. Ekanayake, R. Manohar. Asynchronous DRAM Design and Synthesis. *Proceedings of the 9th IEEE Symposium on Asynchronous Circuits and Systems*, May 2003.

[19] L. Tong, Q. Zhao, and S. Adireddy. Sensor Networks with Mobile Agents. *Proceedings of IEEE Military Communication Conference*, Oct 2003.

[20] C. Kelly, V. N. Ekanayake, R. Manohar. SNAP: A Sensor-Network Asynchronous Processor. *Proceedings of the 9th IEEE Symposium on Asynchronous Circuits and Systems*, May 2003.

[21] J. Kuskin, D. Ofelt, M. Heinrich, et al., The Stanford FLASH Multiprocessor. *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

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

[23] Chipcon CC1000 radio transceiver datasheet. Available: `http://www.chipcon.com/index.cfm?kat_id=2&subkat_id=12&dok_id=14`

[24] CoolRISC Microcontroller Datasheet. Available:`http://www.xemics.com/internet/products/products.jsp?productID=26`

[25] Intel PXA255 XScale Processor Datasheet. Available: `http://www.intel.com/design/pca/prodbref/252780.htm`

[26] Atmel ATMega128L AVR Microcontroller Datasheet. Available: `http://www.atmel.com`

[27] Intel Mote Research Project. Available: `http://www.intel.com/research/exploratory/motes.htm`

[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* . 1998.

[29] A. Mainwaring et al. Wireless Sensor Networks for Habitat Monitoring. *2002 ACM International Workshop on Wireless Sensor Networks and Applications.* Sep 2002.

[30] Wireless Integrated Network Sensors, University of California, Los Angeles, Available: `http://wins.rsc.rockwell.com`

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

[32] C. E. Perkins, E. M. Royer. Ad hoc On-Demand Distance Vector Routing. *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications.* Feb 1999.

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

[34] D. R. Hanson, C. W. Fraser. A Retargetable C Compiler: Design and Implementation. Addison-Wesley, 1995.

[35] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. System architecture directions for network sensors. *ASPLOS 2000.* Nov 2000.

[36] T. D. Burd, T. A. Pering, A. J. Stratakos, R. Brodersen. Dynamic Voltage Scaled Microprocessor System. *IEEE Journal of Solid-State Circuits*, vol. 35, pp. 1571-1580, Nov. 2000.

[37] S. B. Furber, D. A. Edwards and J. D. Garside. AMULET3: a 100 MIPS Asynchronous Embedded Processor. *ICCD'00.* 17-20th September 2000.

[38] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *ARVLSI: Decennial Caltech Conference on VLSI,* ed. C.L. Seitz, 351-373, MIT Press, 1989.

[39] R. Manohar, M. Nyström, A. J. Martin. Precise Exceptions in Asynchronous Processors. *Proceedings of the 19th Conference on Advanced Resesarch in VLSI.* 2001.

[40] J. Tierno, R. Manohar, A.J. Martin. The Energy and Entropy of VLSI Computations. *Proceedings of the 2nd International Conference on Advanced Research in Asynchronous Circuits and Systems*, pp. 188–196, March 1996.

[41] TinyOS Tutorial. `http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/tutorial/index.html`

[42] N. Lee, P. Levis, J. Hill. Mica High Speed Radio Stack. `http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/stack.pdf`. September 2002.

[43] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2003.

[44] B.A. Warneke, K.S.J. Pister. An Ultra-Low Energy Microcontroller for Smart Dust Wireless Sensor Networks. *International Solid-State Circuits Conf.*, February 2004.

[45] B.A. Warneke, et al., Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer Magazine*, Jan 2001.