ULSNAP: An Ultra-Low Power Event-Driven Microcontroller for Sensor Network Nodes

Carlos Tadeo Ortga Otero, Jonathan Tse, Robert Karmazin, Benjamin Hill, and Rajit Manohar
Computer Systems Laboratory, Cornell University
Ithaca, NY, 14853, U.S.A.
{cto3, jon, rob, ben, rajit} at csl.cornell.edu

***Abstract*—We present the architecture of and measured results for ULSNAP: a fully-implemented ultra-low power event-driven microcontroller targeted at the bursty workloads of the sensor network application space. ULSNAP is event-driven at both the microarchitectural and circuit levels in order to minimize static power, energy per operation, and wake up energy while maximizing performance. Our 90 nm test chip offers 93 MIPS at 1.2 V and 47 MIPS at 0.95 V, consuming 47 pJ and 29 pJ per operation respectively. Compared to state-of-the-art processors in its class, ULSNAP is on the Pareto-optimal front of the energy-performance space.**

***Keywords*—VLSI, Low power electronics, Microprocessors, Microcontrollers, Ubiquitous Computing**

I. Introduction

Typical sensor networks are comprised of many small, low cost nodes or "motes" that gather, process, and propagate data about their surrounding environment. Mote deployment lifetimes can exceed several months, making battery life a crucial metric in this design space. Fortunately, most sensor network applications are bursty, e.g. only engaging in active execution when sensor data is available and then returning to a quiescent state. This idle or "sleep" state is often significantly longer than the execution period, so minimizing power during this idle phase is of paramount importance. On the other hand, increasing application complexity requires greater computational power, forcing more aggressive peak performance targets for sensor nodes. The high cost of wireless communication also contributes to increased demand for performance—computing results locally at a sensor mote is often a better system-level tradeoff than wirelessly transmitting raw data [13].

In order to achieve these goals and fit the bursty computation paradigm, ULSNAP was optimized to be event-driven at both the architectural and circuit levels. We make use of the *quasi delay-insensitive* (QDI) family of self-timed, i.e. asynchronous, circuits [8], which are particularly well-suited for event-based computation as they follow a data-driven computational model.

QDI circuits offer automatic fine-grained activity gating behavior in the absence of events, reducing power consumption when the circuits are idle. Traditional synchronous systems attempt to solve this problem using various clock-gating schemes, which introduce complexity and require timing margins to ensure clock stability—QDI circuits are naturally free of these requirements.

Our processor, the Ultra-Low power Sensor Network Asynchronous Processor (ULSNAP), is targeted at this sensor mote application space. When idle, our chip consumes only 9 µW with leakage power as the only contributor. It also has fast wake up time, transitioning from idle to active in only 6.5 ns. When active, our 90 nm chip delivers 93MIPS at 1.2 V and 47MIPS at 0.95 V using 47 pJ and 29 pJ per cycle, respectively. In both the high performance and the low energy mode, ULSNAP is Pareto-optimal in the energy-performance space relative to other state-of-the-art microcontrollers in its class. In fact, ULSNAP can seamlessly operate at different points on the energy-performance curve by scaling its operating voltage.

As for the organization of this paper, we present a brief overview of QDI circuits and their key properties that enable our microarchitectural design and power savings in Sec. II. Sec. III describes our architectural design choices and optimizations, including an event-driven ISA and a hierarchical bus design. Sec. III also discusses our use of a hybrid circuit design, combining two different QDI logic families that fall at opposite ends of the spectrum of pipeline stage complexity. Our memory design for ULSNAP, covered in Sec. IV, leverages optimizations to the SRAM architecture, banks, and bit cells to reduce power consumption. Sec. V details ULSNAP's decoupled coprocessors. Finally, we present detailed measured results of our fully-implemented chip are outlined in Sec. VI.

II. Data-Driven QDI Design

ULSNAP was built using QDI circuits derived using Martin synthesis [8]. The overall design specification was expressed in the Communicating Hardware Processes (CHP) language. A short summary of CHP can be found in the appendix, but as an illustrative example we discuss how to express a pipeline stage implementing some function f. The stage, S, reads a data token from the input channel, IN, and saves it in local variable x. The data token representing the computed value of the function $f(x)$ is then written to the output channel OUT. In CHP, we represent stage S as follows:

$$S \equiv *[IN?x;\ \ OUT!(f(x))]$$

The semantics of the CHP shown above indicate that data must be read from the input channel *before* the function is evaluated. Note that there is no timing bound on the arrival

This work was supported by the NSF under grant CNS 0834582, and the TRUST center under grant CCF 0424422.

The authors would like to acknowledge Prof Sunil Bhave for his invaluable input.

of the input data. The channels over which these hardware processes communicate and synchronize over are completely delay insensitive, i.e. the correctness of the implementation is independent of the delay on any channel. Furthermore, any CMOS circuits derived from this CHP using Martin's synthesis procedure must also follow this data-driven execution model.

Martin's synthesis procedure decomposes a CHP program into many fine-grained hardware processes operating in parallel. These processes synchronize by communicating tokens over delay-insensitive channels that are implemented using a four-phase handshake protocol. A single-bit communication channel is shown in Fig. 1. Here, the sending process asserts the data 0 line to represent a false token (1), which the receiving process acknowledges (2). The channel then resets (3,4). As can be seen in Fig. 1, sending a true token is quite similar.

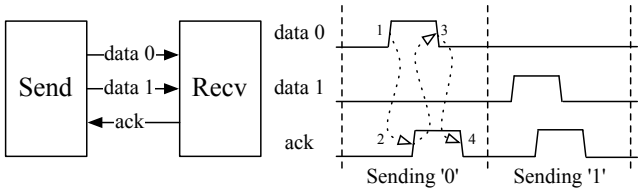


Fig. 1: Four-Phase Handshake

Such a handshake protocol enables local synchronization and flow control on a channel-by-channel basis, which in turn allows designers to optimize for average-case system performance. In contrast to synchronous systems, which must define their clock period by the slowest pipeline stage, the performance of an asynchronous system is set by the critical path of *active* pipeline stages or functional units. The performance of an asynchronous circuit is thus largely governed by the most frequently exercised execution paths, yielding average-case system performance, rather than worst-case performance as in synchronous systems.

This average-case performance property allowed us to optimize rarely used ULSNAP functional units for energy efficiency and frequently used units for performance without the additional design constraints of meeting the strict timing bounds imposed by a global clock. While synchronous designers can implement complex functions as multi-cycle units, they must account for the resultant synchronization and control overheads. Again, in an asynchronous pipeline all synchronization is handled by the local handshakes—there is no additional overhead aside from the momentary reduction in performance when a slow functional unit is exercised.

In addition to being naturally data-driven, QDI circuits operate correctly in the presence of arbitrary wire or gate delays, with the exception that the relative delay on certain wire forks must be bounded. This robustness to delay translates to robustness to variations in the fabrication process, operating voltage, and temperature. By construction, ULSNAP offers a robust computing platform for applications in various environmental conditions.

III. Event-Driven Architecture

While ULSNAP's ISA is fairly standard, the execution model of ULSNAP is event-driven. The initial state of the ULSNAP core is a *wait* state. When an event is triggered, e.g. sensor data arrives, ULSNAP's Event Handler (`EH`) references the Event Register Table (`ERT`), which maps each type of event to a program. The `EH` then initiates the fetch of the appropriate instruction stream for the program indicated by the `ERT` and execution begins. Simultaneous events are handled by arbitration within the `EH`. In some cases it is necessary to trigger an event after some delay. We support this functionality with a timer coprocessor, which contains three decrementing counters—allowing us to delay up to three events. At a count of zero, an event is injected into the event queue and is handled by the `EH`/`ERT`. Specifying a delay time is as simple as initializing a counter to the appropriate value.

Each program is terminated by a `WAIT` instruction, returning the processor to the *wait* state. Thus, ULSNAP is in an idle state when no events are available for processing. Note that there is no explicit power- or clock-gating—idle QDI circuits only consume leakage power in the absence of data.

ULSNAP naturally exploits the data-driven nature of QDI circuits: during a quiescent phase, the underlying circuitry simply waits for data to appear. In such an idle state, no switching activity is present and only leakage power is consumed, achieving a low power envelope. No power management controllers, or clock gating techniques are required to support this behavior. In fact, the `WAIT` instruction is for architectural bookkeeping only. Even a stalled program experiences the benefits of QDI circuits. Only the functional units that can make forward progress have switching activity—inactive or stalled units only consume leakage power. It is important to note that even if the core is in a quiescent state it is ready to compute data—ULSNAP can "wake up" in only 6.5 ns, as detailed in Sec. VI.

A. Microarchitecture

ULSNAP and its predecessor, SNAP [6], implement a 16-bit load-store RISC ISA that supports arithmetic, logic, and branching operations. We have a `gcc`-based toolchain that allows us to compile and execute arbitrary C code on ULSNAP. Instructions are variable length—one to two 16-bit words. A system level diagram of the architecture is shown in Fig. 2. ULSNAP has a more streamlined ISA than SNAP, new I/O and timer coprocessors, and an improved memory architecture.

The processor state in ULSNAP is composed of 16 general purpose registers, a PC register, 4 kB of data memory and 4 kB of instruction memory. The `FETCH` unit addresses the instruction memory and forwards instructions to the `PREDECODE` unit, which then resolves the opcode, source, and destination operands from the incoming instruction stream. All fields of the instruction are passed to the `DECODE` unit, which controls operand flow between the register file (`RFILE`) and all execution units. `DECODE` also controls PC update in the `FETCH`, and any required absolute/relative PC offsets are calculated in the `BRANCH` unit.

In designing the overall microprocessor architecture, we leverage the average-case performance properties of QDI circuits (Sec. II) and divide the execution units in ULSNAP into

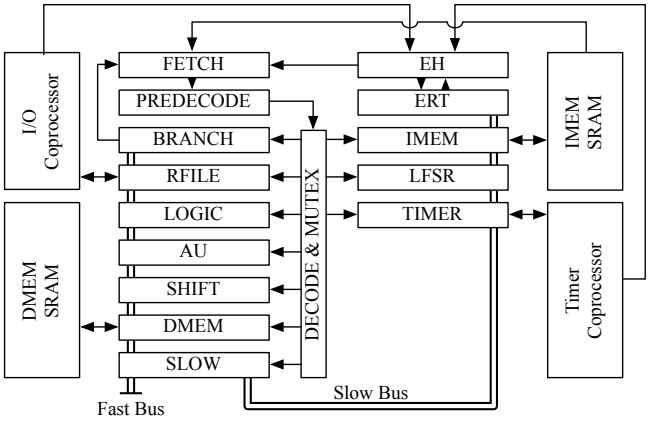


Fig. 2: ULSNAP Architecture

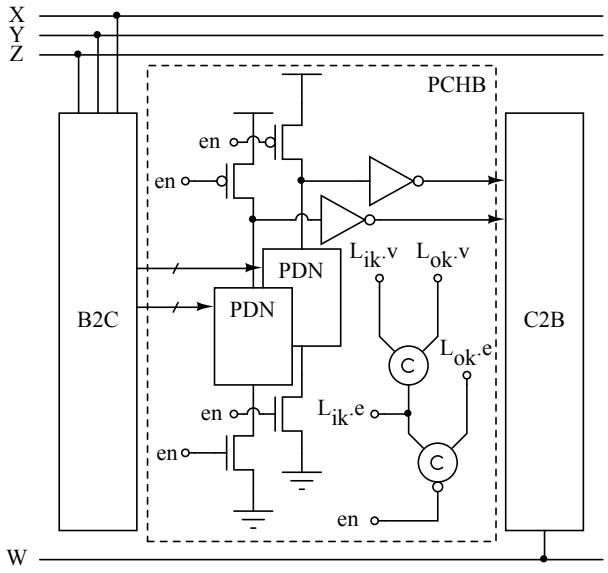
fast and slow groups to improve the overall energy efficiency and performance. Operands and results are transported between execution units and the register file (`RFILE`) by four shared buses: *X* and *Y* for register source operands, *Z* for immediate values, and *W* for results. Frequently used execution units (`RFILE`, `JUMP`, `BRANCH`, `LOGIC`, `ARITH`, `SHIFT`, `DMEM`) are connected directly to these operand and result buses. Less-critical units, i.e. the `LFSR`, `ISTORE`, `TIMER`, and `ERT` units, are decoupled from the buses by a single, dedicated access unit (`SLOW`) as shown in Fig. 2.

This effectively creates two sets of operand and result buses, logically and electrically separating the execution units into fast and slow groups. This has the benefit of significantly shortening the bus wires and reducing per-bus capacitance. For example, we estimate that a monolithic *X* bus would have in excess of 0.4 pF/wire of total capacitance, accounting for both coupling and intrinsic capacitance. Instead the capacitance is split in two segments of 0.17 pF/wire and 0.2 pF/wire for the slow and fast buses respectively. Access to the slow buses incurs an extra overhead of 2 gate delays and an intermediary 0.1 pF/wire.

While the total system capacitance is greater than the estimated monolithic bus capacitance, most of the time the slow units are not accessed. Most operations use only the fast units and therefore only see 0.2 pF. This increased performance for common operations offsets the added latency of access to the slow units and improves overall system performance as we are improving the *average* case execution paths. The non-uniform run times for the execution units poses no synchronization problem since our self-timed methodology is robust to gate and wire delays (Sec. II). We quantify the relative difference between the slow and fast execution paths using a synthetic benchmark, discussed in Sec. VI.

B. Circuit Implementation

We make use of a hybrid approach at the circuit level, combining two different QDI logic families: precharge buffer templates [10] and control-data decomposition. These two families fall at opposite ends of the spectrum of pipeline stage complexity. Precharge buffer pipeline templates, such as PCHB and PCFB, were widely used in the MiniMIPS processor [12]. Each PCHB/PCFB stage typically implements a function of small enough complexity that it can fit into a single nMOS pulldown network, as illustrated in Fig. 3. This compilation style yields high-performance stages with short cycle time. However, a reasonably complex function must be decomposed into a pipeline of several PCHB/PCFB stages, resulting in a long latency for a single data token to travel through the entire pipeline, though maintaining a high token throughput.



Center shows bit slice of the datapath an of execution unit.

Fig. 3: ULSNAP Execution Unit Template

Conversely, control-data decomposition, used in the Caltech Asynchronous Microprocessor [11], typically aggregates computation into a single pipeline stage. While the cycle time of such a stage is higher than the equivalent PCHB/PCFB pipeline, the overall latency and energy consumption are less. Circuits compiled using either of these methods are completely inter-operable, which allows the designer to tailor the latency/cycle time of all computational units individually.

We implemented high throughput execution units such as the `ARITH` and `BRANCH` units using the PCHB/PCFB templates. Fig. 3 depicts an example bit slice of such an execution unit. Given the small amount of computation these units perform, the PCHB/PCFB pipeline is at most 2 stages. This represented a reasonable tradeoff between throughput and energy/latency for these stages. The fetch loop and predecode units are compiled using the control-data technique, which allowed us to minimize the latency of key computations such as updating the PC.

As described earlier, all the functional units are connected by operand (X, Y, and Z) and result (W) buses, each of which is a shared channel. Channels only provide synchronization between a single produce and consumer, i.e. they are not multicast. Some additional hardware is necessary to preserve the local synchronization handshakes described in Sec. II, so we wrapped each unit with bus-to-channel (`B2C`) and channel-to-bus (`C2B`) interfaces, which are controlled by the `DECODE` unit. As an example, we show the `B2C` interface for the X bus

and the C2B interface for the W bus in CHP[1]:

$$
\begin{aligned}
B2C \;&\equiv \\
&*[[\overline{Read_k}];\; x_k{\uparrow};\; Read_k;\; L_{ik}!(X?);\; x_k{\downarrow}] \\
C2B \;&\equiv \\
&*[[\overline{Write_k}];\; x_k{\uparrow};\; Write_k;\; W!(L_{ok}?);\; x_k{\downarrow}]
\end{aligned}
$$

Fig. 3 shows a PCHB-style functional unit with B2C and C2B interfaces. The internal PCHB pipeline stage accepts input on channel L_{ik} and produces outputs on channel L_{ok}. The $Read_k$ and $Write_k$ are dataless channels connecting the DECODE unit to each of the B2C and C2B interfaces. We have expanded the above CHP descriptions to include an internal state variable x_k, which we discuss later. By interfacing with the appropriate $Read_k$ or $Write_k$ channel(s), the decode unit can guarantee each functional unit mutually exclusive access to the appropriate operand and result buses.

Unlike all other functional units, the DECODE unit is not implemented with PCHB/PCFB or control-data style pipeline templates, as it must provide resource allocation functionality. To address this specific need we make use of Pipelined Mutual Exclusion (PME) [9]. In short, by using PME the DECODE unit synchronizes the fetch and execution units while allowing the fetch loop to continue execution as the execution unit(s) processes the previous instruction. This simple optimization allows us to introduce concurrency with little overhead.

PME can be described as follows: given a set of mutually exclusive actions $(A_1, A_2, \ldots, A_n)$ and a command channel to execute each of those actions $(C_1, C_2, \ldots, C_n)$ we can guarantee mutual exclusion by implementing the following CHP program:

$$
\begin{aligned}
p_i \;&\equiv \\
&*[[\overline{S_i}];\; x_i{\uparrow};\; S_i;\; A_i;\; x_i{\downarrow}] \;\;\|\\
&*[[\overline{C_i} \wedge (\forall j : j \neq i : \neg x_j)];\; S_i;\; C_i]
\end{aligned}
$$

Note that the above CHP program consists of two separate programs running in parallel, synchronized by synchronization channels $S_1, S_2, \ldots, S_n$. The first process of p_i is structurally identical to that of the B2C and C2B processes, which essentially replace A_i with the appropriate channel actions on the bus and local channels. In the PME context, the set of state variables $x_0, x_1, ..., x_n$ behave as a distributed synchronization lock that reserves resources when executing an action, e.g. the shared bus channels.

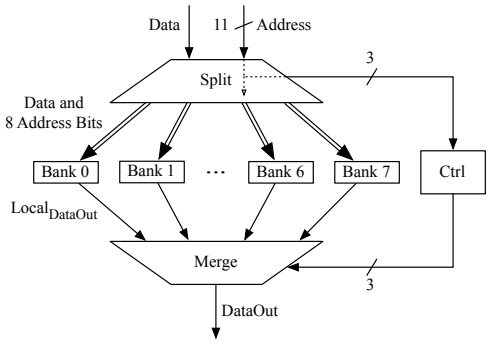
The key feature of PME is how it allows a control process communicating on the command channels C_j, in this case the decode logic, to continue execution without waiting for the commanded action A_j to complete. To illustrate this, let us assume that the action A_k is desired, and that it is the first action, i.e. there are no current actions being performed. The controlling process initiates a channel action on C_k. The wait condition $[\overline{C_k} \wedge (\forall j : j \neq k : \neg x_j)]$ is met, so a channel action on S_k is initiated. The next wait condition $[\overline{S_k}]$ is now met, reserving the shared resource by raising x_k. At this time, the channel actions on S_k and C_k are allowed to complete, freeing the controlling process to continue work.

[1]A short summary of CHP can be found in the Appendix

Synchronization happens when the DECODE unit tries to execute the next action by initiating the appropriate command channel action on C_i. At this point, the controller must wait for all locks x_j to become **false**, which in our example will only occur once A_k has finished. By decoupling the DECODE unit from function unit action completions, we can begin decoding an instruction while the previous instruction is being executed, adding additional concurrency to our execution.

IV. Memory Architecture

ULSNAP has 8 kB of memory, divided equally into instruction and data memory. Both memories are organized into 8 banks as shown in Fig 4. A memory operation is handled by a SPLIT process that addresses the correct bank using the least significant bits of the address. The SPLIT process was compiled using a full buffer reshuffling (PCFB) that allows multiple outstanding operations—up to one per bank. Read operations make use of the MERGE process, which selects the appropriate $\mathrm{Local}_{\mathrm{DataOut}}$ bus and ships the result back to the core. In the case of contiguous memory access or small strides, we can leverage our ability to have multiple outstanding memory operations to different banks. In this way the PCFB reshuffling enables us to reduce performance requirements for each bank without starving the processor core.



Banks are addressed using LSB bits of the address. Only active banks consume dynamic power during a memory request.

Fig. 4: SRAM Organization

In order to further reduce static power, the SRAM bit cell relies on long channel devices to reduce leakage. The total SRAM leakage is 4 µW for all 8 kB of memory. For reference, a direct port of the original 180 nm SNAP memory to our more modern 90 nm process consumes more than 200 µW of leakage power. Note that this reduction of static power between designs does not come at the cost of a significant latency increase. In fact, due to the multiple outstanding requests enabled by our use of PCFB-reshuffled logic, the average SRAM access latency of ULSNAP is similar to that of a single cycle of the microcontroller core.

Each bank is divided into 64 rows and 4 columns, each of which is 4 B (2 words) wide. We chose this configuration to allow for relatively short bit lines. Shortening the bit lines reduces switching capacitance and improves noise margins. Reads from the SRAM are fully QDI, since a read operation will eventually cause a bit line transition which can be

detected. However, we cannot observe the state of a write operation by only inspecting the bit lines. In order to provide a timing bound for a write operation, we build a delay-line-like structure out of a dummy SRAM column, placed on the side of the SRAM farthest from the word line drivers. During a write operation on the SRAM, we perform a read on this dummy column and wait for its transition to be detected.

The placement of the dummy column at the end of the word line accounts for the maximum possible delay on the word line. Furthermore, since this dummy column is identical in every other respect to actual SRAM columns, the bit line capacitance charge/discharge timing characteristics are comparable. The key assumption is that reads take longer than writes, padding our delay margins. This configuration allows us to have a dummy delay-replica-loop that approximates the delays associated with the physical design as well as global and systemic process variations.

V. Coprocessors

A. Timer Coprocessor

Providing efficient hardware support to schedule events in the future is crucial in order to maximize the amount of time the ULSNAP core can remain idle, using only leakage power. To this end, both SNAP and ULSNAP implement timer coprocessors. The timer coprocessor in ULSNAP is composed of three 24-bit decrementing counters or "timers." Each counter can be independently initialized to a positive integer through the use of two custom instructions in the ULSNAP ISA: `SCHEDHI` and `SCHEDLO`. These instructions set the most and least significant bits of each counter, respectively. When a timer expires, i.e. the counter has been decremented to 0 from the initial value, the timer coprocessor injects an event into the Event Handler (`EH`) event queue.

The original SNAP timer coprocessor was constructed from a single always-running clock or "tick generator" and three decrementing counters. Gating the clock signal connection to each of the counters enabled or disabled each of the counters, providing three controllable timers. While simple to implement, this approach did not leverage a key benefit of asynchronous circuits: intrinsic activity gating. The use of a continuously running clock is power inefficient, especially when considering the required distribution to three counters.

ULSNAP makes an improvement to this design by implementing per-timer stoppable clocks for each of the three counters. This enables per-timer activity gating and reduces the amount of global wiring. Furthermore, the frequency of each of the clocks is configurable, allowing for different wait times and wait time precision. Each of the timers is completely decoupled from the others providing significant savings in power consumption.

A detailed picture of each timer can be seen in Fig. 5. Each timer has a tick generator (Tic Gen) that generates tokens on a dataless asynchronous channel (dec) at a user-configurable frequency. The dec channel serves as the command to decrement the counter. The ULSNAP core configures, sets, starts, stops, and resets each timer via the $ctrl$ channel. As timer commands can arrive even if the timer is active, especially if the timer is being reconfigured or reset, the $ctrl$ and dec channels must be

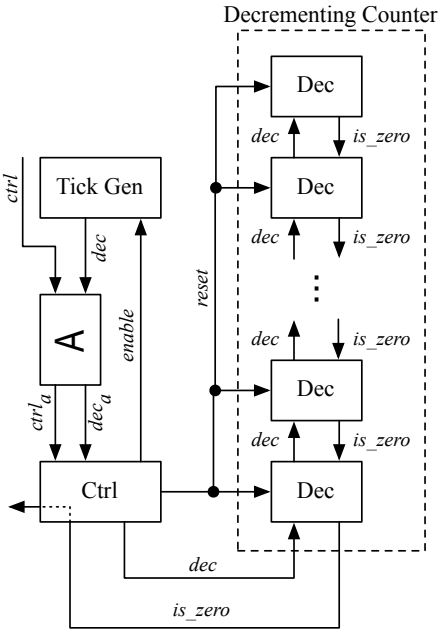


Fig. 5: Timer Implementation

arbitrated. The resulting $ctrl_a$ and dec_a channels are mutually exclusive. The Ctrl process is responsible for initializing the decrementing counter, enabling/disabling the Tic Gen process, as well as detecting when the counter is zero and injecting an event into the Event Handler.

The decrement counter Fig. 5 is implemented as a serial pipeline of n-bit decrementer processes (Dec), each of of which corresponds to a single bit of counter. The Ctrl process can reset and initialize each unit via a $reset$ channel, which is connected to each decrementer process. Each Dec process stores two variables: b, the actual value of the counter bit, and z, a bookkeeping value which is true if all bits more significant than the current position are zero. The z value enables two key features. The first is to minimize the number of exercised Dec units while decrementing—if all higher order bits are zero, there is no need to interact with them thus saving power. Secondly, because each Dec unit has information about itself as well the higher bits relative to itself, the decrement action is constant response time, similar to the empty pipeline detection counter of [15]. If a decrement operation must borrow from a more significant bit, a channel action takes place on the dec channel. The next Dec process responds with the appropriate z value on the is_zer_o channel to keep all the state updated. As there is no clock, a constant-time asynchronous cycle is of great importance in a decrementing counter used as a timer.

B. I/O coprocessor

Off-chip communication is handled by an I/O coprocessor. In comparison to SNAP, ULSNAP's I/O coprocessor has been made more modular, enabling easy support of different serial protocols. Currently, we implement two off-chip serial protocols in the I/O coprocessor: SPI, and a simple asynchronous serial protocol similar to that shown in Fig. 1. Communication between the I/O coprocessor and the core is done through an

I/O-mapped register (R15). Whenever an I/O event occurs, an token is placed into the Event Handler queue and the associated data is pushed into the input queue of R15.

The SPI unit can only be used in master mode. The frequency of the SPI clock can be configured through an off-chip delay line. We support all SPI clock polarity (CPOL) and phase (CPHA) modes, each of which can be configured by initializing the SPICFG register in the I/O coprocessor to the appropriate values. In order to avoid a race condition between writing to the SPICFG register and the transmit (SPITX) or receive (SPIRX) registers, the I/O coprocessor inserts an event into the Event Handler's event queue whenever the configuration is changed. The SPI unit can be configured in transmit, receive or duplex mode. To preserve the event-driven architectural model, the I/O coprocessor will inject an event into the Event Handler queue whenever a word is received through the SPI or serial unit. The throughput of the serial asynchronous interface of the I/O coprocessor is a 16 bit serial message every 2 cycles (130 ns). The throughput of the serial I/O interface is limited by our padframe design and the capacitance of the PCB traces.

VI. Evaluation

ULSNAP was fabricated in a commercial 90 nm low-power CMOS process using a full-custom layout flow. The processor core alone contains 122k transistors in an area of $0.312\,\text{mm}^2$. Including the memories, the transistor count is 592k in an area of $0.844\,\text{mm}^2$. *All* reported power measurements include the memory power consumption. A photo of the die and development board is shown in Fig. 6. ULSNAP is the successor to the SNAP processor and is fabricated in a more modern 90 nm process. When appropriate, we make the relevant comparisons between a simulated 90 nm ULSNAP and our ULSNAP design.

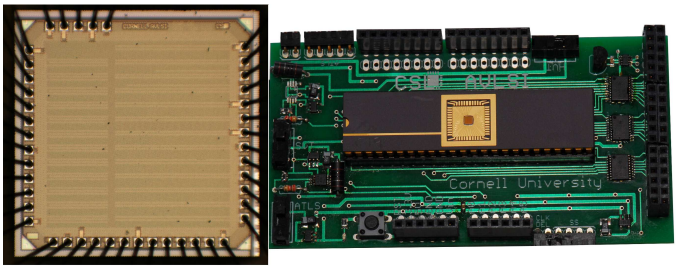


Fig. 6: Die photo and development board

To evaluate the static power consumption, we tested 10 separate ULSNAP chips with empty event queues, i.e. there was no activity. Since QDI circuits provide automatic fine-grained activity gating, and our design does not contain any busy loops or waits in the absence of events or instructions, leakage power is the only source of power consumption while idle. Note that this is not an explicit power-gated state and that there are no explicit hardware power management structures in ULSNAP—although extending the design to include them is possible [15]. Fig. 7 presents measured static power consumption from all leakage paths, and also shows how static power scales with V_{DD}.

To evaluate the processing performance of ULSNAP when active, we developed micro-benchmarks that stress the processor with ALU and memory operations. We measured the performance and energy for our micro-benchmarks while varying the supply voltage from 1.2 V to 0.95 V. As ULSNAP has no clock, we have no direct control over the operating frequency save for changing V_{DD}. Note that this voltage and "frequency" scaling is a natural benefit to our use of QDI circuits and requires no explicit hardware support or design effort.

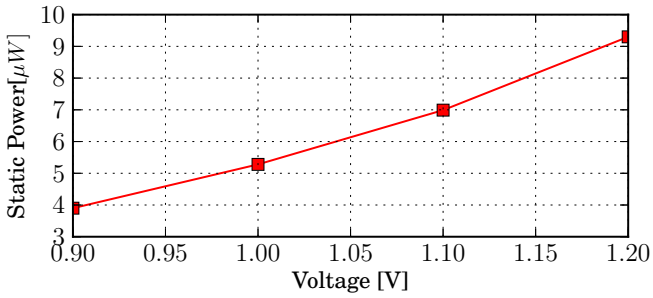


Fig. 7: Static power consumption

We report power and performance characteristics in Fig. 8. Our 90 nm test chip delivers an average of 93MIPS at 1.2 V using 47 pJ per cycle. These numbers are an average of measurements across 17 different ULSNAP cores with standard deviation 5 MIPS and 0.5 pJ. ULSNAP can be also run in low energy mode with V_{DD} at 0.95 V. In this mode, it only uses 29 pJ per cycle while delivering 47MIPS of integer operations.

Fig. 8 also shows that ULSNAP automatically adjusts to multiple voltages, allowing it to operate at different points on the energy-performance curve. A smart application could control the supply voltage to ULSNAP using a digitally controlled power source/regulator to optimize the energy-performance trade off during the lifetime of the sensor mote. While this approach is possible on synchronous circuits, it requires a focused design effort to close timing.

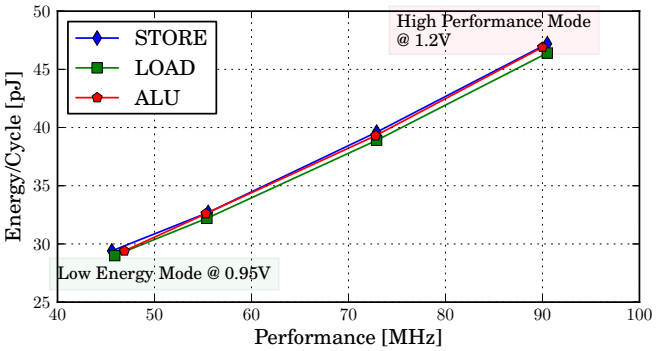


Fig. 8: Performance-energy trade off

We also include an evaluation of six standard benchmarks for embedded processors [14] implemented in C and compiled to our custom ISA with our gcc flow. Performance and energy measurements at 1.2 V for each benchmark are shown in Table I.

Finally, we developed a micro-benchmark to stress the timer coprocessor with the main core idle—this is possible

as the three timers are decoupled from the main core. Our measurements show that each timer can reach average frequencies up to 270 MHz while consuming only 0.85 pJ/cycle/timer. While each timer's individual frequency is configurable, in our testing we ran them all at the same frequency. When idle, ULSNAP's timer coprocessor uses only 300 nW at nominal V_{DD}, as compared to 400 µW for the coprocessor in [6].

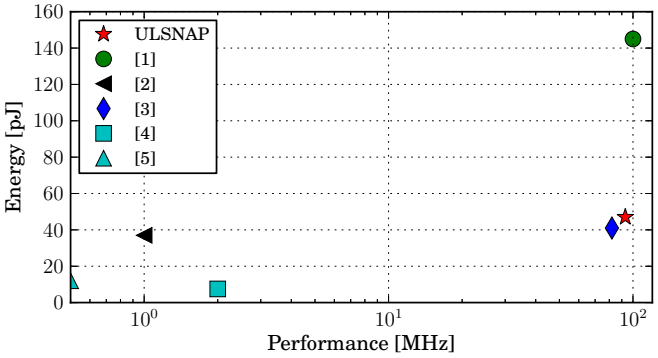


Fig. 9: Energy-performance comparison of processors in high perf. mode

We compared ULSNAP against various state-of-the-art microcontrollers [1-5] and present the results in Table II. Fig. 9 and Fig. 10 shows that ULSNAP is Pareto-optimal in the energy-performance space in both low-energy and high performance modes relative to all other state-of-the-art microcontrollers. In other words, ULSNAP is superior in either performance or energy, if not both metrics, as compared to other deeply embedded microcontrollers. This is achieved by a combination of factors: ULSNAP's event-driven design, microarchitectural optimizations such as bus partitioning, and circuit implementation details such as the use of self-timed circuits. As power and performance numbers are workload dependent, the numbers reported in Table II and Fig 9 are for the workload that performs best on each microcontroller.

To measure the effect of splitting the datapath buses into two fast and slow buses, we performed an experiment that performs 20×10^6 consecutive memory *writes* first to the Data memory `DMEM` and then to the Instruction memory `ISTORE`. The unit responsible to access `DMEM` is connected to the fast buses while the unit responsible to write into the Instruction memory is connected to the slow buses as shown in Fig 2. While this is not an exhaustive test of every functional unit on both buses, the `DMEM` and `ISTORE` interfaces are identical save that they are connected to different buses. Thus this test is representative of the overheads in accessing the slow bus.

Writes to the data memory completed in 270 ms (15 ns/access), while writes to the instruction memory were completed in 526 ms (29 ns/access). This is consistent with our expectations since the `ISTORE` interface uses the slow buses while the `DMEM` interface is connected to the fast buses as shown in Fig. 2. The performance disparity between `DMEM` and `ISTORE` is a good indicator that bus splitting improves the performance of commonly executed instructions at the expense of rarely used instructions.

TABLE I: Benchmarks

Task	Perf [tasks/s]	E [nJ/task]	Input
CRC4	3.19×10^5	12	16b
Tiny Encryption (TEA)	8.41×10^3	490	64b(data)
Int Average	6.37×10^3	652	2kB
MinMax	4.73×10^3	821	2kB
Search	1.55×10^3	27	2kB
Serial RX	1.63×10^3	7	16b

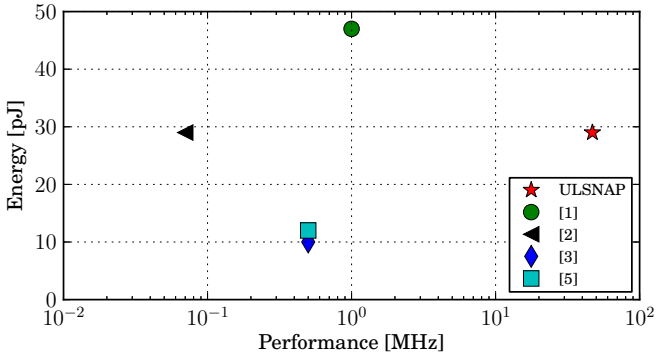


Fig. 10: Energy-performance comparison of processors in low energy mode

TABLE II: Comparison of State of the Art Microcontrollers

	[1]	[2]	[3]	[4]	[5]	ULSNAP
Tech [nm]	90	180	65	180	250	90
Datapath [bits]	24	32	32	8	8	16
SRAM [kB]	2000	3	16	0.33	3.12	8
High Performance Mode						
Supply [V]	1.2	0.5	1.2	0.9	1.0	1.2
Perf. [MHz]	100	1	82	2	0.5	93
Energy [pJ]	145	37	41	7.5	12	47
Low Energy Mode						
Supply [V]	0.4	0.4	0.5	0.5	NA	0.95
Perf. [MHz]	1	0.07	0.5	0.1	NA	47
Energy [pJ]	47	29	10	2.8	NA	29

Reported numbers for High Performance Mode are for minimum cycle time workloads. Low Energy mode numbers are for workloads which minimize energy.

The time between event arrival and ULSNAP reacting, i.e. waking up from idle, is 6.5 ns. Upon receiving an external message or a timer event, full control is transferred to the core from the Event Handler within 14.8 ns. The first instruction starts execution within 40 ns. Note that these latencies are from SPICE simulation of extracted layout with full parasitics, since they are not directly observable on our test setup.

Fig. 11 shows the power envelope of an encryption benchmark (TEA) that is representative of the benefits of ULSNAP's event-driven design. This benchmark receives (Rx) 16 kB data from the serial interface, encrypts the data, and transmits (Tx) the result over the serial interface. During the Tx and Rx phases the power consumption is only 22 µW. When all the data is available, encryption runs at full throughput (93 MHz). After transmission it naturally goes into a deep sleep mode and uses only 9 µW. The average power consumption of ULSNAP on this benchmark is only 98 µW.

Note that the TEA benchmark is not annotated with power or sleep directives. In fact, the programmer need not explicitly

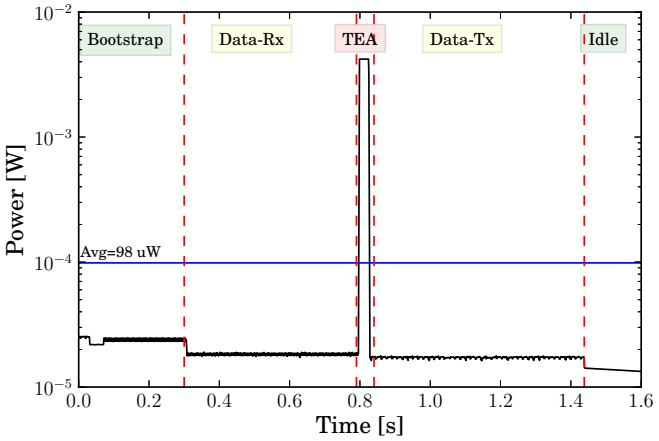


Fig. 11: Power profile of an encryption benchmark (TEA) [14]

define a sleep mode at all. The trace in Fig. 11 illustrates that ULSNAP will automatically scale power usage with activity.

VII. Conclusion

ULSNAP's core achieves state-of-the-art energy and performance thanks to optimizations at the architecture, microarchitecture, and circuit levels. Its event-driven architecture matches the low power, bursty performance requirements of sensor network applications.

Microarchitectural choices such as our hierarchical slow/fast bus design further reduce power and improve average case performance. We maximized the energy efficiency of our memory by banking each memory module and decoupling the buses, activating only accessed banks. Furthermore, we allowed multiple outstanding memory operations by reshuffling the memory access modules. We chose long-channel memory bit cells to minimize power consumption of idle memory locations. Additionally, ULSNAP's timer and I/O coprocessors were optimized to minimize the power in the quiescent state by, among other things, decoupling their control from the core.

Finally, our self-timed circuits are event-driven without any additional control overheads. QDI circuits minimize power consumption during idle periods and automatically adjust to variations in voltage and other environmental factors.

We have presented *measured* results for ULSNAP: a fully implemented ultra-low power event-driven microcontroller offering high performance within a low energy envelope. With respect to state-of-the-art processors in its class, ULSNAP offers Pareto-optimal operating points in the energy-performance space. It achieves 93MIPS at 1.2 V and 47MIPS at 0.95 V while using 47 pJ and 29 pJ per cycle, respectively.

ULSNAP is a good fit for sensor network motes with bursty, computationally intensive workloads. It dynamically scales its throughput to maintain the lowest possible power envelope at all times without any programmer effort.

Appendix

The CHP notation we use is based on Hoare's CSP [7]. A full description of CHP and its semantics can be found in [8]. What follows is a short and informal description.

- Assignment: $a := b$. $a\uparrow$ is shorthand for $a := true$, and $a\downarrow$ for $a := false$.
- Selection: $[G1 \rightarrow S1 \,[]\, ... \,[]\, Gn \rightarrow Sn]$. G_i are boolean expressions (guards) and S_i are program parts. Execution stalls until a Gi is true, at which point Si is executed. $[G]$ is short-hand for $[G \rightarrow \text{skip}]$, which stalls until $G = true$.
- Repetition: $*[G1 \rightarrow S1 \,[]\, ... \,[]\, Gn \rightarrow Sn]$. Choose $Gi = true$, execute Si. Repeat until no Gi is true. $*[S]$ is short-hand for $*[true \rightarrow S]$.
- Send: $X!e$. Evalute expression e and send result over channel X.
- Receive: $Y?v$. Receive value over channel Y and store variable v.
- Probe: $\overline{X}$ is a boolean which is $true$ if and only if a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.
- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

References

[1] Ashouei, M.; Hulzink, J.; Van Ginderdeuren, J.; et al, *"A voltage-scalable biomedical signal processor running ECG using 13pJ/cycle at 1Mhz and 0.4V"*, ISSCC, 2011.

[2] Chen, G.; Fojtik, M.; Blaauw, D.; et al, *"Millimeter-scale nearly perpetual sensor system with stacked battery and solar cells"*, ISSCC, 2010

[3] Ickes, N.; Sinagil, Y.; Pappalardo, F.; Guidetti, E.; Chandrakasan, A., *"A 10 pJ/cycle ultra-low-voltage 32-bit microprocessor system-on-chip"*, ESSCIRC, 2011

[4] Mingoo, S.; Hanson, S.; Blaaw, D. et al, *"The Phoenix Processor: A 30pW platform for sensor applications"*, Symposium on VLSI, 2008

[5] Warneke, B.; Pister, K.S.J., *"An ultra-low energy microcontroller for Smart Dust wireless sensor networks"*, ISSCC, 2004

[6] Ekanayake, V.; Kelly, C.; Manohar, R., *"An ultra low-power processor for sensor networks"*, ASPLOS, 2004

[7] Hoare, C.A.R., *"Communicating Sequential Processes"*, CACM, 1978

[8] Martin, A., *"Compiling Communicating Processes into Delay-Insensitive VLSI Circuits"*, Distributed Computing, 1986

[9] Manohar, R; Martin, A., *"Pipelined Mutual Exclusion and The Design of an Asynchronous Microprocessor"*, Computer Systems Laboratory Technical Report, CSL-TR-2001-1017, 2001

[10] Lines, A., *"Pipelined Asynchronous Circuits"*, MS Thesis, Caltech, 1995

[11] Martin, A.; Burns, S.; et al, *"The design of an asynchronous microprocessor"*, SIGARCH Computer Architecture, 1989

[12] Martin, A.; Lines, A.; Manohar, R.; et al, *"The design of an asynchronous MIPS R3000 microprocessor"*, Advanced Research in VLSI, 1997

[13] Akyildiz, I.F.; Su, W. et al, *Wireless sensor networks: a survey* Computer Networks, Elsevier, 2002

[14] Nazhandali, L.; Minuth, M.; Austin, T., *"SenseBench: toward an accurate evaluation of sensor network processors"*, Workload Characterization Symposium, 2005

[15] Ortega, C.; Tse, J.; Manohar, R., *"Static Power Reduction Techniques for Asynchronous Circuits"*, ASYNC 2010