ISSN: 2277-9655
Impact Factor: 1.852

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

DSP Architecture for Wireless Sensor Nodes Using VLSI Technique

R.Ragumadhavan
Assistant Professor, Department of ECE, PSNA College Of Engineering And Technology, India
ragufriends@gmail.com

***Abstract*— Radio communication exhibits the highest energy consumption in wireless sensor nodes. Given their limited energy supply from batteries or scavenging, these nodes must trade data communication for on-the-node computation. Currently, they are designed around off-the-shelf low-power microcontrollers. But by employing a more appropriate processing element, the energy consumption can be significantly reduced. This paper describes the design and implementation of the newly proposed folded-tree architecture for on-the-node data processing in wireless sensor networks, using parallel prefix operations and data locality in hardware. Measurements of the silicon implementation show an improvement of 10–20× in terms of energy as compared to traditional modern micro-controllers found in sensor nodes.**

***keywords*—Digital processor, parallel prefix, wireless sensor network (WSN).**

I. INTRODUCTION

WIRELESS sensor network (WSN) applications range from medical monitoring to environmental sensing, industrial inspection, and military surveillance. WSN nodes essentially consist of sensors, a radio, and a microcontroller combined with a limited power supply, e.g., battery or energy scavenging. Since radio transmissions are very expensive in terms of energy, they must be kept to a minimum in order to extend node lifetime. The ratio of communication-to-computation energy cost can range from 100 to 3000 [1]. So data communication must be traded for on-the-node processing which in turn can convert the many sensor readings into a few useful data values. The data-driven nature of WSN applications requires a specific data processing approach. Previously, we have shown how parallel prefix computations can be a common denominator of many WSN data processing algorithms [2]. The goal of this paper is to design an ultra-low-energy WSN digital signal processor by further exploiting this and other characteristics unique to WSNs.

First, Section II lists these specific characteristics of WSN-related on-the-node processing. Then, Section III covers the proposed approach to exploit these properties. Section IV elaborates on the programming and usage of the resulting folded tree architecture. Section V discusses the application-specific integrated circuit (ASIC) implementation of the design while Section VI measures its performance and Section VII illustrates the usefulness to WSNs with four relevant case- studies. Finally, the work is concluded in Section VIII.

II. CHARACTERISTICS OF WSNS AND RELATED REQUIREMENTS FOR PROCESSING

Several specific characteristics, unique to WSNs, need to be considered when designing a data processor architecture for WSNs.

Data-Driven: WSN applications are all about sensing data in an environment and translating this into useful information for the end-user. So virtually all WSN applications are characterized by local processing of the sensed data [3].

Many-to-Few: Since radio transmissions are very expensive in terms of energy, they must be kept to a minimum in order to extend node lifetime. Data communication must be traded for on-the-node computation to save energy, so many sensor readings can be reduced to a few useful data values.

Application-Specific: A "one-size-fits-all" solution does not exist since a general purpose processor is far too power hungry for the sensor node's limited energy budget. ASICs, on the other hand, are more energy efficient but lack the flexibility to facilitate many different applications.

Apart from the above characteristics of WSNs, two key requirements for improving existing processing and control architectures can be identified.

Minimize Memory Access: Modern micro-controllers (MCU) are based on the principles of a divide-and-conquer strategy of ultra-fast processors on the one hand and arbitrary complex programs on the other hand [4]. But due to this generic approach, algorithms are deemed to spend up to 40–60% of the time in accessing memory [5], making it a bottle-neck [6]. In addition, the lack of task-specific operations leads to inefficient execution, which results in longer algorithms and significant memory book keeping.

Combine Data Flow and Control Flow Principles: To manage the data stream (to/from data memory) and the instruction stream (from program memory) in the core functional unit, two approaches exist. Under control flow, the data stream is a consequence of the instruction stream, while under data flow the instruction stream is a consequence of the data stream. A traditional processor architecture is a control flow machine, with programs that execute sequentially as a stream of instructions. In contrast, a data flow program identifies the data dependencies, which enable the processor to more or less choose the order of execution. The latter approach has been hugely successful in specialized high-throughput applications, such as multimedia and graphics processing. This paper shows how a combination of both approaches can lead to a significant improvement over traditional WSN data processing solutions.

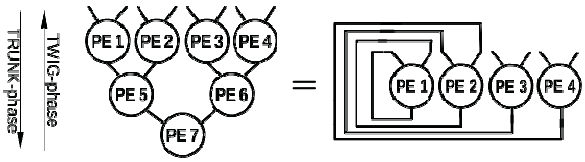


Fig. 1. A binary tree (left, 7 PEs) is functionally equivalent to the novel folded tree topology (right, 4 PEs) used in this architecture.

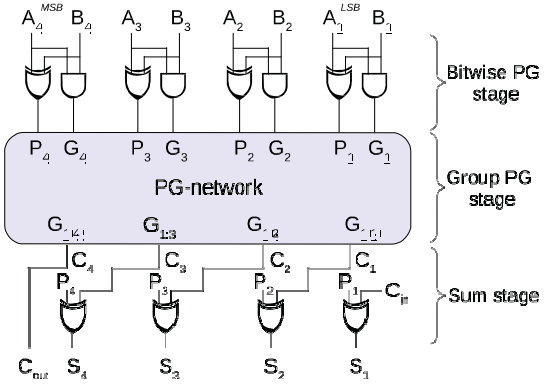


Fig. 2. Addition with propagate-generate (PG) logic.

III. Proposed Approach

A. WSN Applications and On-The-Node Data Aggregation

Notwithstanding the seemingly vast nature of WSN applications, a set of basic building blocks for on-the-node processing can be identified. Common on-the-node operations performed on input data collected directly from the node's sensors or through in-the-network aggregation include filtering, fitting, sorting, and searching [7]. We published earlier [2] that these types of algorithms can be expressed in terms of parallel prefix operations as a common denominator.

Prefix operations can be calculated in a number of ways [8], but we chose the binary tree approach [9] because its flow matches the desired on-the-node data aggregation. This can be visualized as a binary tree of processing elements (PEs) across which input data flows from the leaves to the root (Fig. 1, left). This topology will form the fixed part of our approach, but in order to serve multiple applications, flexibility is also required. The tree-based data flow will, therefore, be executed on a data path of programmable PEs, which provides this flexibility together with the parallel prefix concept.

B. Parallel Prefix Operations

In the digital design world, prefix operations are best known for their application in the class of carry look-ahead adders [10]. The addition of two inputs A and B in this case consists of three stages (Fig. 2): a bitwise propagate-generate (PG) logic stage, a group PG logic stage, and a sum-stage.

The outputs of the bitwise PG stage ($P_i = A_i \oplus B_i$ and $G_i = A_i \cdot B_i$) are fed as (P_i, G_i)-pairs to the group PG logic stage, which implements the following expression:

$$(P_i, G_i) \circ (P_{i+1}, G_{i+1}) = (P_i \cdot P_{i+1}, G_i + P_i \cdot G_{i+1}) \qquad (1)$$

It can be shown this $\circ$-operator has an identify element $I = (1, 0)$ and is associative.

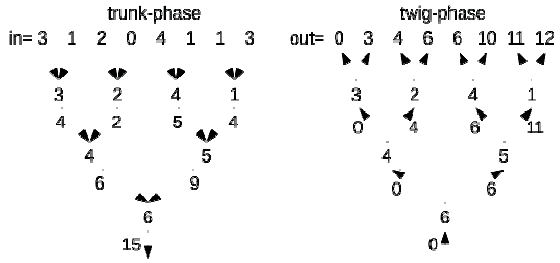


Fig. 3. Example of a prefix calculation with sum-operator using Blelloch's generic approach in a trunk- and twig-phase.

For example, the binary numbers A = "1001" and B = "0101" are added together. The bitwise PG logic of LSB-first noted $A = \{1001\}$ and $B = \{1010\}$ returns the PG-pairs for these values, namely, $(P, G) = \{(0, 1); (0, 0); (1, 0); (1, 0)\}$. Using these pairs as input for the group PG-network, defined by the $\circ$-operator from (1) to calculate the prefix operation, results in the carry-array $G = \{1, 0, 0, 0\}$ [i.e., the second element of each resulting pair from (1)]. In fact, it contains all the carries of the addition, hence the name carry look-ahead. Combined with the corresponding propagate values P_i, this yields the sum $S = \{0111\}$, which corresponds to "1110."

The group PG logic is an example of a parallel prefix computation with the given $\circ$-operator. The output of this parallel-prefix PG-network is called the all-prefix set defined next.

Given a binary closed and associative operator $\circ$ with identity element I and an ordered set of n elements $[a_0, a_1, a_2, \ldots, a_{n-1}]$, the reduced-prefix set is the ordered set $[I, a_0, (a_0 \circ a_1), \ldots, (a_0 \circ a_1 \circ \cdots \circ a_{n-2})]$, while the all-prefix set is the ordered set $[a_0, (a_0 \circ a_1), \ldots, (a_0 \circ a_1 \circ \cdots \circ a_{n-1})]$, of which the last element $(a_0 \circ a_1 \circ \cdots \circ a_{n-1})$ is called the prefix element.

For example, if $\circ$ is a simple addition, then the prefix element of the ordered set $[3, 1, 2, 0, 4, 1, 1, 3]$ is $L, a_i = 15$. Blelloch's procedure [9] to calculate the prefix-operations on a binary tree requires two phases (Fig. 3). In the trunk-phase, the left value L is saved locally as Lsave and it is added to the right value R, which is passed on toward the root. This continues until the parallel-prefix element 15 is found at the root. Note that each time, a store-and-calculate operation is executed. Then the twig-phase starts, during which data moves in the opposite direction, from the root to the leaves. Now the incoming value, beginning with the sum identity element 0 at the root, is passed to the left child, while it is also added to the previously saved Lsave and passed to the right child. In the end, the reduced-prefix set is found at the leaves.

An example application of the parallel-prefix operation with the sum operator (prefix-sum) is filtering an array so that all elements that do not meet certain criteria are filtered out. This is accomplished by first deriving a "keep"-array, holding "1" if an element matches the criteria and "0" if it should be left out. Calculating the prefix-sum of this array will return the amount as well as the position of the to-be-kept elements of the input array. The result array simply takes an element from the input array if the corresponding keep-array element is "1" and copies it to the position found in the corresponding element

of the prefix-sum-array. To further illustrate this, suppose the criterion is to only keep odd elements in the array and throw away all even elements. This criterion can be formulated as keep$(x) = (x \bmod 2)$. The rest is calculated as follows:

$$\begin{aligned} \text{input} &= [2, 3, 8, 7, 6, 2, 1, 5] \\ \text{keep} &= [0, 1, 0, 1, 0, 0, 1, 1] \\ \text{prefix} &= [0, 1, 1, 2, 2, 2, 3, 4] \\ \text{result} &= [3, 7, 1, 5]. \end{aligned}$$

The keep-array provides the result of the criterion. Then the parallel-prefix with sum-operator is calculated, which results in the prefix-array. Its last element indicates how many elements are to be kept (i.e., 4). Whenever the keep-array holds a "1," the corresponding input-element is copied in the result-array at the index given by the corresponding prefix-element (i.e., 3 to position 1, 7 to position 2, etc.). This is a very generic approach that can be used in combination with more complex criteria as well.

Other possible applications that relate to WSNs include peak detection, polynomial evaluation for model-fitting, lexically compare strings, add multi-precision numbers, delete marked elements from arrays, and quick sort [3], [11]. Some of these will be used as a case-study later in Section VII.

C. Folded Tree

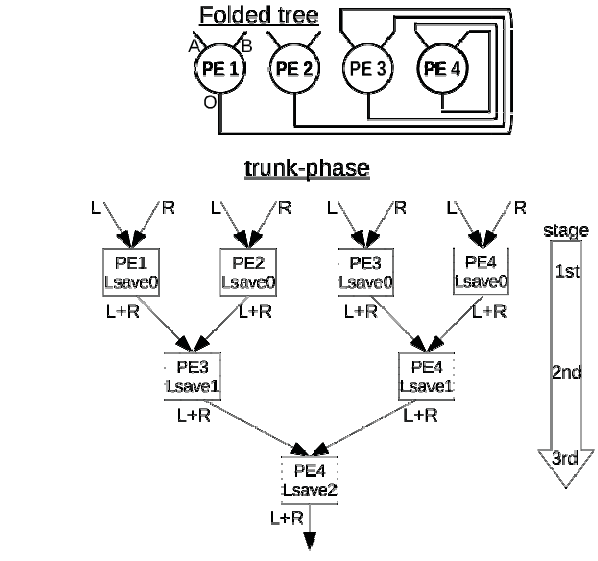
However, a straightforward binary tree implementation of Blelloch's approach as shown in Fig. 3 costs a significant amount of area as n inputs require $p = n - 1$ PEs. To reduce area and power, pipelining can be traded for throughput [8]. With a classic binary tree, as soon as a layer of PEs finishes processing, the results are passed on and new calculations can already recommence independently.

The idea presented here is to fold the tree back onto itself to maximally reuse the PEs. In doing so, p becomes proportional to $n/2$ and the area is cut in half. Note that also the interconnect is reduced. On the other hand, throughput decreases by a factor of $\log_2(n)$ but since the sample rate of different physical phenomena relevant for WSNs does not exceed 100 kHz [12], this leaves enough room for this tradeoff to be made. This newly proposed folded tree topology is depicted in Fig. 1 on the right, which is functionally equivalent to the binary tree on the left.

IV. Programming and Using the Folded Tree

Now it will be shown how Blelloch's generic approach for an arbitrary parallel prefix operator can be programmed to run on the folded tree. As an example, the sum-operator is used to implement a parallel-prefix sum operation on a 4-PE folded tree.

First, the trunk-phase is considered. At the top of Fig. 4, a folded tree with four PEs is drawn of which PE3 and PE4 are hatched differently. The functional equivalent binary tree in the center again shows how data moves from leaves to root during the trunk-phase. It is annotated with the letters L and R to indicate the left and right input value of inputs A and B. In accordance with Blelloch's approach, L is saved as Lsave and the sum L+R is passed. Note that these annotations are not global, meaning that annotations with the same name do not necessarily share the same actual value.



		Write RF		Read RF		
Label	description	A	B	A	B	Operation
0: T	L → Lsave0 & L + R → O	inputA @ 0b00	inputB @ 0b00	0b00 @ Lsave0	0b00 @ R	ADD A+B
1: T	L → Lsave1 & L + R → O	inputA @ 0b01	inputB @ 0b01	0b01 @ Lsave1	0b01 @ R	ADD A+B
2: T	L → Lsave2 & L + R → O	inputA @ 0b10	inputB @ 0b10	0b10 @ Lsave2	0b10 @ R	ADD A+B

Fig. 4. Implications of using a folded tree (four4-PE folded tree shown at the top): some PEs must keep multiple Lsave's (center). Bottom: the trunk-phase program code of the prefix-sum algorithm on a 4-PE folded tree.

To see exactly how the folded tree functionally becomes a binary tree, all nodes of the binary tree (center of Fig. 4) are assigned numbers that correspond to the PE (1 through 4), which will act like that node at that stage. As can be seen, PE1 and PE2 are only used once, PE3 is used twice and PE4 is used three times. This corresponds to a decreasing number of active PEs while progressing from stage to stage. The first stage has all four PEs active. The second stage has two active PEs: PE3 and PE4. The third and last stage has only one active PE: PE4. More importantly, it can also be seen that PE3 and PE4 have to store multiple Lsave values. PE4 must keep three: Lsave0 through Lsave2, while PE3 keeps two: Lsave0 and Lsave1. PE1 and PE2 each only keep one: Lsave0. This has implications toward the code implementation of the trunk-phase on the folded tree as shown next.

The PE program for the prefix-sum trunk-phase is given at the bottom of Fig. 4. The description column shows how data is stored or moves, while the actual operation is given in the last column. The write/read register files (RF) columns show how incoming data is saved/retrieved in local RF, e.g., $X@0bY$ means X is saved at address $0bY$, while $0bY@X$ loads the value at $0bY$ into X. Details of the PE data path (Fig. 8) and the trigger handshaking, which can make PEs wait for new input data (indicated by **T**), are given in Section V. The trunk-phase PE program here has three instructions, which are identical, apart from the different RF addresses that are used. Due to the fact that multiple Lsave's have to be stored, each stage will have its own RF address to store and retrieve them.

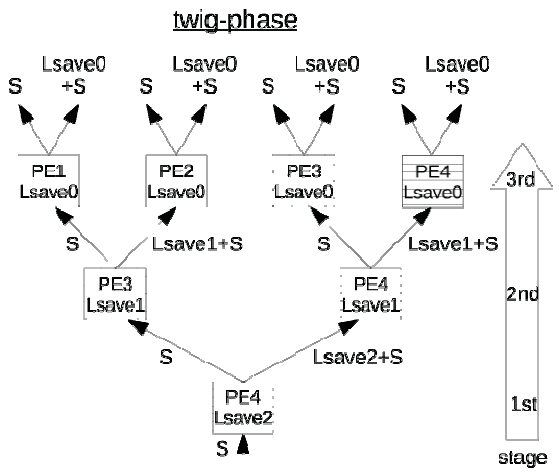


Fig. 5. Annotated twig-phase graph of 4-PE folded tree.

This is why PE4 (active for 3 stages) needs three instructions (lines ⓪-②), PE3 (active for 2 stages) needs two instructions (lines ⓪-①) and PE1 and PE2 (active in first stage only) need one instruction (line ⓪). This basically means that the folding of the tree is traded for the unrolling of the program code.

Now, the twig-phase is considered using Fig. 5. The tree operates in the opposite direction, so an incoming value (annotated as S) enters the PE through its O port [see Fig. 4(top)]. Following Blelloch's approach, S is passed to the left and the sum S + Lsave is passed to the right. Note that here as well none of these annotations are global. The way the PEs are activated during the twig-phase again influences how the programming of the folded tree must happen. To explain this, Fig. 6 shows each stage of the twig-phase (as shown in Fig. 5) separately to better see how each PE is activated during the twig-phase and for how many stages. The annotations on the graph wires (circled numbers) relate to the instruction lines of the program code shown in Fig. 7, which will also be discussed.

Fig. 6 (top) shows that PE4 is active during all three stages of the twig-phase. First, an incoming value (in this case the identity element S2) is passed to the left. Then it is added to the previously (from the trunk-phase) stored Lsave2 value and passed to the right. PE4-instruction ① will both pass the sum Lsave2 + S2 = S1 to the right (= itself) and pass this S1 also the left toward PE3. The same applies for the next instruction ②. The last instruction ③ passes the sum Lsave0+S0.

Looking at the PE3 activity [Fig. 6 (center)], it is only active in the second and third stage of the twig-phase. It is indeed only triggered after the first stage when PE4 passes S2 to the left. The first PE3-instruction ⓪ passes S2 to PE1, and instruction ① adds this to the saved Lsave1, passing this sum T1 to PE2. The same procedure is repeated for the incoming S1 from PE4 to PE3, which is passed to its left (instruction ②), while the sum Lsave0+S1 is passed to its right (instruction ③). In fact, two pairs of instructions can be identified, that exhibit the same behavior in terms of its outputs: the instruction-pair ⓪ and ① and the instruction-pair ② and ③. Two things are different however. First, the used register addresses (e.g., to

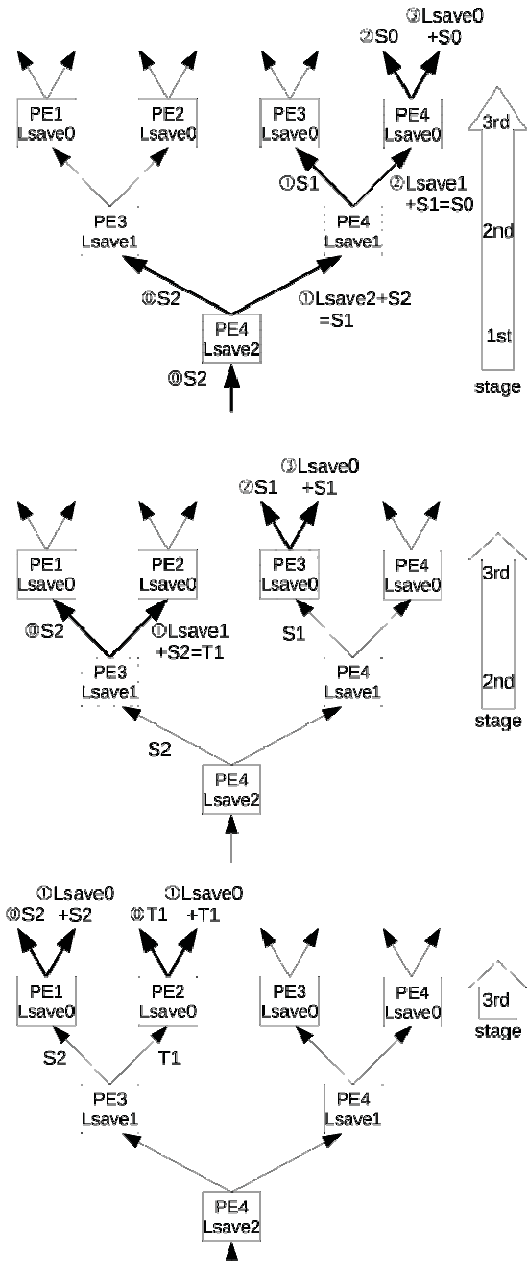


Fig. 6. Activity of different PEs during the stages of the twig-phase for a 4-PE folded tree: PE4 (top), PE3 (center), PE1 and PE2 (bottom).

store Lsave values) are different. Second, the first pair stores incoming values S0 and S1 from PE4, while the second pair does not store anything. These differences due to the folding, again lead to unrolled program code for PE3.

Last, PE1 and PE2 activity are shown at the bottom of Fig. 6. They each execute two instructions. First, the incoming value is passed to the left, followed by passing the sum of this value with Lsave0 to the right. The program code for both is shown in the bottom two tables of Fig. 7.

V. Hardware Implementation

Fig. 8(a) gives a schematic overview of the implemented folded tree design. The ASIC comprises of eight identical 16-bit PEs (Fig. 10), each consisting of a data path with

PE 4	TWIG description	Write RF A	Write RF B	Read RF A	Read RF B	Operation	to output
0 T	Identity = S2 → Left	-	-	-	0b11 @ identity	PASS B	A
1	(Lsave2+S2=S1) → L&R	-	previous @ 0b11 = S2	0b10 @ Lsave2	0b11 @ S2	ADD A+B	A and B
2	(Lsave1+S1=S0) → L&R	-	previous @ 0b11 = S1	0b01 @ Lsave1	0b11 @ S1	ADD A+B	A and B
3	(Lsave0+S0) → Right	-	previous @ 0b11 = S0	0b00 @ Lsave0	0b11 @ S0	ADD A+B	B

PE 3	TWIG description	Write RF A	Write RF B	Read RF A	Read RF B	Operation	to output
0 T	S2 → Left	-	inputO @ 0b11 = S2	-	0b11 @ S2	PASS B	A
1	(Lsave1+S2=T1) → Right	-	inputO @ 0b10 = S1	0b01 @ Lsave1	0b11 @ S2	ADD A+B	B
2	S1 → Left	-	-	-	0b10 @ S1	PASS B	A
3	(Lsave0+S1) → Right	-	-	0b00 @ Lsave0	0b10 @ S1	ADD A+B	B

PE 2	TWIG description	Write RF A	Write RF B	Read RF A	Read RF B	Operation	to output
0 T	S2 → Left	-	inputO @ 0b01 = S2	-	0b01 @ S2	PASS B	A
1	(Lsave0+S2) → Right	-	-	0b00 @ Lsave0	0b01 @ S2	ADD A+B	B

PE 1	TWIG description	Write RF A	Write RF B	Read RF A	Read RF B	Operation	to output
0 T	T1 → Left	-	inputO @ 0b01 = T1	-	0b01 @ T1	PASS B	A
1	(Lsave0+T1) → Right	-	-	0b00 @ Lsave0	0b01 @ T1	ADD A+B	B

Fig. 7. Program of the twig-phase of the prefix sum algorithm for a 4-PE folded tree.

programmable controller and 16 × 36 bit instruction memory. They are interconnected through the request-acknowledge handshaking trigger bus and the bidirectional data bus in the folded way (cf. Fig. 1). Handshaking triggers activate the PEs only when new data is available and in such a way that they functionally become a binary tree in both directions of the trunk- and twig-phase. Within each data path [Fig. 8(b)], muxes select external data, stored data or the previous result as the next input for the data path. The data path contains an algorithmic logical unit (ALU) with four-word deep register files (RF-A and RF-B) at the inputs A and B for operand isolation. These RFs comprise the distributed data memory of the whole system, with a combined capacity of 4 kB. They are the only clocked elements within the data path. As data flows through the tree, it is constantly kept local to its designated operation. This is one of the goals of this paper, which effectively removes the von Neumann bottleneck and

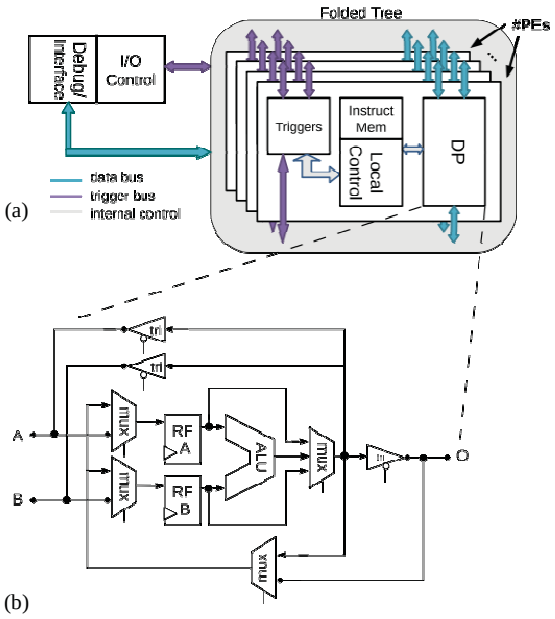


Fig. 8. Schematic diagram of design overview. (a) Top-level view (here with four PEs shown). (b) Detail of one PE data path
.

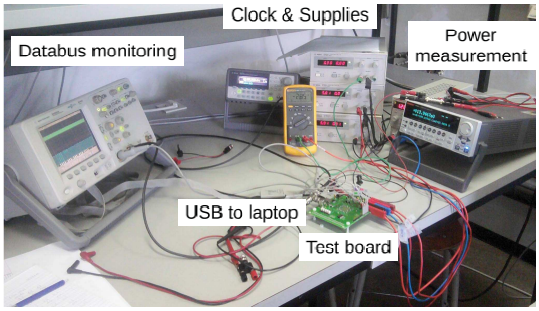


Fig. 9. View of the lab measurement setup for the folded tree IC.

saves power. The design targets 20–80-MHz operation at 1.2 V. It was fabricated in 130-nm standard cell CMOS.

A PE takes six (down-phase) or seven (up-phase) cycles to process one 36-bit instruction, which can be divided into three stages.

1) Preparation, which acknowledges the data and starts the core when input triggers are received (1 cycle).
2) Execution, which performs the load-execute-jump stages to do the calculations and fetch the next instruction pointer (4 cycles).
3) Transfer, which forwards the result by triggering the next PE in the folded tree path on a request-acknowledge basis (1–2 cycle).

This is tailored toward executing the key store-and-calculate operation of the parallel prefix algorithm on a tree as described earlier in Section III-B. Combined with the flexibility to program the PEs using any combination of operators available in their data path, the folded tree has the freedom to run a variety of parallel-prefix applications [11].

VI. Experimental Validation

The measurement setup (Fig. 9) of the chip uses digital interfacing over universal serial bus (USB) to access the data I/O, programming, and debug facilities. The data bus [see

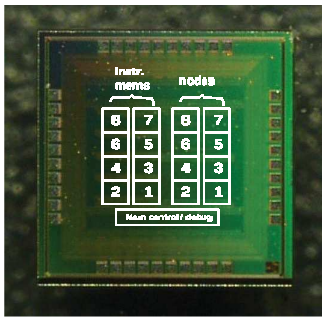


Fig. 10. Die photograph of the implemented processor with eight PEs.

TABLE I
LEAKAGE POWER AND DYNAMIC ENERGY FOR ONE PE UNDER NOMINAL CONDITIONS (20 MHz, 1.2 V)

1 Processing Element	Active PE core	Idle PE core	PE Instr. Mem.
Dynamic energy/instr (pJ)	14.6	4.7	2.10
Leakage power (µW)	0.03	0.03	0.01
Total power @ 20 MHz (µW)	41.7	13.5	6.0

Fig. 8(a)] activity can be monitored and plotted, as will be shown in Section VII.

A. PE Measurements

The PE table in Table I gives the dynamic energy and leakage power for one PE core running at 20 MHz and 1.2 V supply under full stress with varying data inputs. It consumes 42 μW or 2.1 μW/MHz, including 0.03 μW leakage. The register-based instruction memory power values are presented in last column of the PE table and consume 6 μW. When going into Idle mode, a PE will consume 60% less than in Active mode.

To validate the measurements and to check whether the derived values for a single PE are correct, they are combined in an estimate for the folded tree design with eight PEs. When such a folded tree executes a trunk-phase, it will take four stages to reach the root. Thanks to the handshaking, at each stage, the number of active PEs is cut in half as 8, 4, 2, 1. The number of idle PEs increases accordingly as 0, 4, 6, 7. This makes a total of 15 active PEs and 17 idle PEs. By combining this information with the PE's consumption, the folded tree consumption can be estimated. As can be seen in the table of Table II, this closely matches the measured values for the folded tree. The last column also takes the instruction memories into account. Overall, the folded tree processor consumes 255 μW or 13 pJ/cycle, including memories.

B. Energy-Per-Instruction

A standard benchmark suite of applications for WSN systems does not exist though some initial attempts have been made [17], [7]. Without running the same applications on each platform, it is not possible to fairly compare energy efficiency, performance, and flexibility. This is especially true for academic results, which all revert to different benchmarks due to the lack of a standard suite. As a consequence, readers

TABLE II
FOLDED TREE CIRCUIT WITH EIGHT PES EXECUTING A TRUNK-PHASE UNDER NOMINAL CONDITIONS (20 MHz, 1.2 V)

Folded Tree (trunk-phase)	Estimate	Measured	incl.Instr. Mem.
Total dynamic energy (pJ)	298.8	289.2	356.4
Leakage power (µW)	0.25	0.25	0.35
Total power @ 20 MHz (µW)	213.7	206.8	254.9

TABLE III
ENERGY PER INSTRUCTION OF RELATED WORK, NORMALIZED TO 130 nm, 1.2 V, AND 16 BIT [13]–[16]

Core	Arch	DP (bits)	Process (nm)	VDD (V)	Clk (MHz)	E/instr (pJ)	Norm. E/Instr (pJ)
SNAP/LE	RISC GP	16	180	1.8	200	218.0	157.4
Hempstead	RISC GP	8	130	0.6	12	3.9	30.4
BitSNAP	RISC GP	16	180	1.8	54	54.0	17.3
Smart Dust	RISC GP	8	250	1.0	0.5	12.0	18.0
one single PE	CISC GP	16	130	1.2	20	4.2	4.2

are often left with only the energy-per-instruction metric to compare different systems. Table III presents a summary of related academic work. The listed energy-per-instruction values are normalized to the presented work using following formula:

$$E_{\text{norm}} = E_{\text{orig}} \times 130\ \text{nm}/L \times (1.2\ V/V_{\text{dd}})^2 \times 16\ \text{bit}/W \quad (2)$$

given energy per instruction E_{orig}, process L, supply V_{dd}, and data path bitwidth W of the other system. This work requires at least 4.3× less in terms of energy per instruction. The notion of an instruction, however, might significantly differ especially as WSN systems often employ specific instruction sets and specialized hardware to reach extreme energy efficiency. This is the case for this work as well since the benefit of the parallel prefix-sums framework cannot be fully quantified using the small-scale energy-per-instruction metric.

C. Algorithmic Unit

A better metric for comparison is the energy per algorithmic unit (AU). The AU is a sequence of frequently used steps in the target applications. To calculate this metric, a complete data sheet with full instruction set and detailed power measurements is needed. In contrast to academic work, this information is readily available for many commercial MCUs.

Given the context of WSN applications, the AU is defined as a load-execute-store-jump sequence, which is a key in data processing algorithms that loop over data arrays. For each MCU, the total number of cycles for the AU sequence is calculated. Each time, the most efficient instructions are chosen from each MCU's specific instruction set. The energy per cycle is based on the information found in the data sheet and normalized using (2). Table IV presents the details of this comparison. The OpenMSP 430 [18], which is an open-source model of the widely-used MSP430 MCU, is also included. It has been taken through sign-off P&R for accurate power simulation results. A single PE outperforms other MCUs by at least 20× in terms of energy, requiring only 2.4 pJ per cycle or 16.8 pJ per AU at equal clock speed.

To correctly compare the MCUs with the eight PEs in the folded tree, the parallel aspect of the latter needs to be taken

TABLE IV

COMPARING TOTAL ENERGY FOR THE ALGORITHMIC UNIT (AU) SEQUENCE (LOAD-EXECUTE-STORE-JUMP)

Core	Arch	DP (bits)	Memory (KB)	Process (nm)	Supply Current (µA)	VDD (V)	Clk (MHz)	Energy/cycle (pJ)	Normalized E/cyc (pJ)	Algo Unit (cycles)	Algo Unit Norm. E (pJ)
ATxmega128D4	RISC GP	8	8	250	1100	3.0	2	1650	274.6	8	2196
STM8L101	CISC GP	8	1.5	130	900	3.0	8	338	108.0	8	864
TI MSP430F550x	RISC GP	16	4	180	2840	3.0	20	426	49.2	8	394
OpenMSP430	RISC GP	16	2	130	765	1.2	20	45.9	45.9	8	367
This work – 1 PE	CISC GP	16	0.5	130	40	1.2	20	2.4	2.4	7	17
This work – Tree	CISC Tree	16	4	130	214	1.2	20	12.9	12.9	7	90

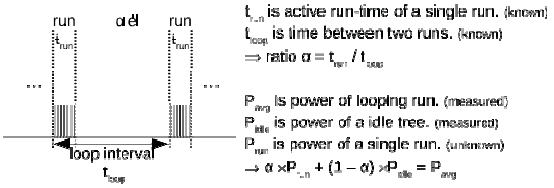


Fig. 11. Time definitions for loop-runs.

into account. As derived earlier, a folded tree of eight PEs will execute 15 AU's over four stages or 3.75 on average. In other words, the folded tree must be compared to the equivalent of 3.75 MCUs. The folded tree then outperforms the closest competitor MSP430 by at least 15× in terms of energy.

VII. CASE-STUDIES OF EXAMPLE ALGORITHMS

Finally, despite the lack of standardized benchmark algorithms, a selection of four relevant example algorithms is made. Each of these algorithms will be introduced and measured for their performance in terms of energy consumption and speed. The result is compared to the performance of the OpenMSP430. Based on the previous experiments, this is the closest competitor. The most efficient MSP430-instructions are again used.

With regards to the implemented ASIC (Fig. 10), a strategy must be developed for measuring the correct energy consumption. Simply executing an algorithm on the folded tree once is too fast to measure anything useful. So alternatively, the same algorithm can be executed multiple times. To derive a correct energy value, the loop overhead time must be taken into account. This can be accomplished by also measuring the idle consumption and calculating the ratio of active algorithm time versus overall time, including loop overhead.

The activity of a looping run of an algorithm is represented in Fig. 11. When an algorithm is looping, it will be active for a time t_{run} during which it will consume an amount of power P_{run}. The run time t_{run} can be directly derived from the number of instructions in the program code. The time between two runs within the loop t_{loop} is also known. This time is significant, since for this measurement the folded tree is controlled in MATLAB by a UART-over-USB interface of 78 125 baud or 128 μs between restarts. The ratio of these two, $\alpha = t_{\text{run}}/t_{\text{loop}}$, returns the percentage of algorithm activity with respect to the total loop time. The inverse ratio is the actual loop time overhead percentage.

The average power consumption of the looped algorithm, P_{avg}, can be measured. Also, the idle consumption of the tree, P_{idle}, can be measured. The sought P_{run} can then be calculated

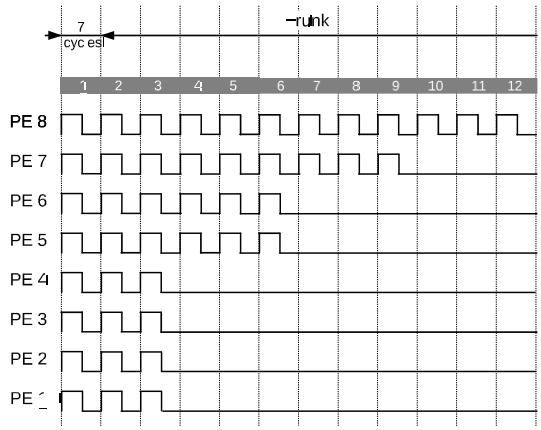


Fig. 12. Peak detection and polynomial evaluation algorithm PE activity.

from the expression

$$\alpha \cdot P_{\text{run}} + (1 - \alpha) \cdot P_{\text{idle}} = P_{\text{avg}}. \quad (3)$$

The different times can be calculated (Fig. 11) based on the algorithm program code and checked against the activity of the data bus. Examples of such activity plots will be shown in the following sections along with the measurements of the example algorithms.

A. Peak Detection

WSN nodes are often involved in operational modes that only activate when a certain trigger level is reached over a set of readings during an amount of time, e.g., to control room heating. The operator needed for finding the maximum is $a \circ b = (a > b \ ? \ a : b)$.

Fig. 12 presents the PE activity of a single run on the folded tree chip for this algorithm. It shows eight traces in which each "1" represents the completion of an instruction by the corresponding PE. Fig. 12 shows the four stages during the trunk-phase of an eight-PE folded tree, corresponding to 8, 4, 2, and 1 active PEs.

The complete algorithm takes t_{run} = 4 groups × 3 instr × 7 cycl/instr (trunk-mode) × 50 ns/cycl (@ 20 MHz) = 4200 ns or α = 3.3% which, together with the measured P_{avg} = 146 μW, leads to E_{run} = 986 pJ. Including the instruction memory, the total energy consumption is E_{total} = 1265 pJ, compared to 9538 pJ for the MSP430.

B. Evaluate Polynomial

Various polynomial models exist to fit sensor data in order to decide whether this data is useful or not and whether any

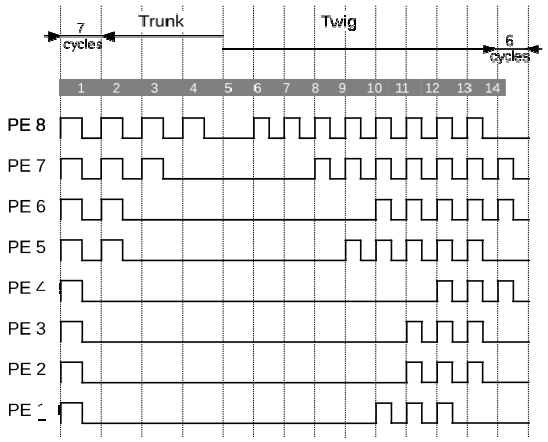


Fig. 13. All-prefix sum algorithm PE activity.

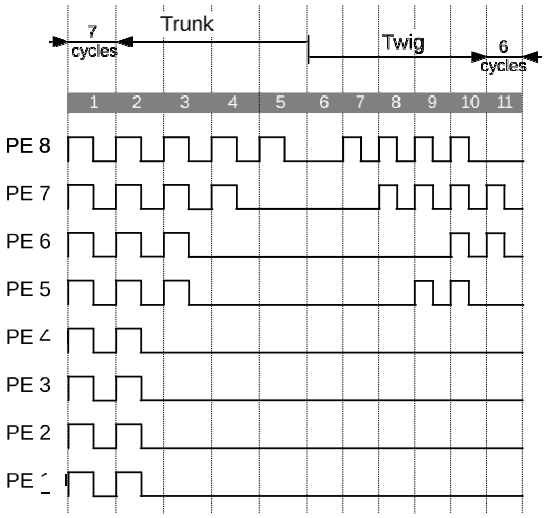


Fig. 14. Find elements algorithm PE activity.

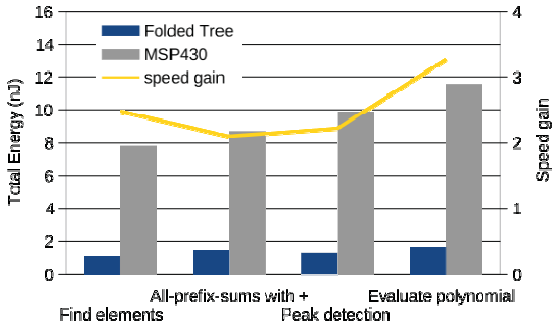


Fig. 15. Total energy consumption of example algorithms (20 MHz, 1.2 V).

action should be undertaken. [19] shows how a $(n-1)$-th order polynomial $a_n + a_{n-1}x^1 + \cdots + a_1 x^{n-1}$ can be evaluated with parallel-prefix operations by pairing the coefficients together with the desired evaluation value x.

The same Fig. 12 can be used in this case as well. Although the program code and structure differs from the previous algorithm, they both execute a trunk-phase in four stages of three instructions. The complete algorithm takes t_{run} = 4 groups * 3 instr * 7 cycl/instr (trunk-mode) * 50 ns/cycl (@ 20 MHz) = 4200 ns or α = 3.3% which, together with the measured P_{avg} = 150 μW, leads to E_{run} = 1431 pJ. Including the instruction memory, the total energy consumption is E_{total} = 1697 pJ, compared to 11 289 pJ for the MSP430.

C. All-Prefix Sum

Fig. 13 presents the PE activity of the all-prefix sum algorithm described earlier in Section III-B. Again, the four stages during the trunk-phase of a eight-PE folded tree can be seen, corresponding to 8, 4, 2, and 1 active PEs. After the transition to twig-phase, the different triggering of the different PEs can be deduced. For example, the twig-phase starts at the root PE8 with passing the identity element to the left, where PE7 is indeed the first one to start after PE8.

The complete algorithm takes t_{run} = [4 instr × 7 cycl/instr (trunk-mode) + (1+9) instr * 6 cycl/instr (twig-mode)] * 50 ns/cycl (@ 20 MHz) = 4400 ns or α = 3.4% which, together with the measured P_{avg} = 147 μW, leads to E_{run} = 1183 pJ. Including the instruction memory, the total energy consumption is E_{total} = 1492 pJ, compared to 8319 pJ for the MSP430.

D. Find Elements in Array

The flag function needed for finding matching elements between two arrays is $E(a, b) = (a = b\ ?\ 1 : 0)$. Comparable to the selection example under Section III-B, the prefix-sum of this array can be used to retrieve the index of the matched elements [11].

Fig. 14 presents the PE activity of a single run on the folded tree chip of this algorithm. In the first two steps, all PEs execute two instructions to compare all 2 × 8 elements against the searched value and pass a “1” if matched, “0” otherwise. The resulting array of eight elements is then taken through a parallel prefix sum operation. In other words, a prefix-sum essentially using a 4-PE folded tree. Here, the activity and triggering of the different PEs can be deduced and corresponds to what was shown earlier in Figs. 4 and 6. The complete algorithm takes t_{run} = [4 instr * 5 cycl/instr (trunk-mode) + (1+5) instr * 6 cycl/instr (twig-mode)] * 50 ns/cycl (@ 20 MHz) = 3550 ns or α = 2.7% which, together with the measured P_{avg} = 146 μW, leads to E_{run} = 908 pJ. Including the instruction memory, the total energy consumption is E_{total} = 1140 pJ, compared to 7974 pJ for the MSP430.

E. Results

All case-study results are summarized in Table V and 15. The folded tree outperforms the MSP430 by 8–10× in terms of energy and at least 2–3× in terms of execution time. Note that this speed gain can be traded for even more energy-efficient execution by lowering the supply voltage until an equal throughput is reached. Operating at half the frequency (10 MHz) and a minimal supply voltage of 0.79 V, the processor consumes about half the energy. A single active PE core will now only consume 0.95 μW/MHz, including leakage. Overall, the folded tree processor now consumes down to 80 μW or 8 pJ/cycle and running the example algorithms,

TABLE V

TOTAL ENERGY CONSUMPTION OF EXAMPLE ALGORITHMS (20 MHz, 1.2 V)

	for Folded Tree Chip					for MSP430		
Algorithm (1.2V @ 20MHz)	Run-time (ns)	alpha T_{loop}=128us	E_{nodes} (pJ)	$E_{instr.mem.}$ (pJ)	E_{TOTAL} (pJ)	Run-time (ns) @ 20 MHz	#cycli	$E_{running}$ (pJ)
Find elements	3550	2.77%	908	232	1140	8800	176	7974
All-prefix sum	4400	3.44%	1183	309	1492	9250	185	8319
Peak detection	4200	3.28%	986	279	1265	10100	202	9538
Evaluate polynomial	4200	3.28%	1431	266	1697	13750	275	11289

it outperforms other MCUs by at least 20× in terms of total energy. Finally, the folded tree chip was combined with a radio and sensor on a prototype sensor node. Measurements indicated that using the proposed architecture significantly reduces radio communication and, in a typical WSN application, can save up to 70% of the total sensor node energy.

VIII. CONCLUSION

This paper presented the folded tree architecture of a digital signal processor for WSN applications. The design exploits the fact that many data processing algorithms for WSN applications can be described using parallel-prefix operations, introducing the much needed flexibility. Energy is saved thanks to the following: 1) limiting the data set by pre-processing with parallel-prefix operations; 2) the reuse of the binary tree as a folded tree; and 3) the combination of data flow and control flow elements to introduce a local distributed memory, which removes the memory bottleneck while retaining sufficient flexibility.

The simplicity of the programmable PEs that constitute the folded tree network resulted in high integration, fast cycle time, and lower power consumption. Finally, measurements of a 130-nm silicon implementation of the 16-bit folded tree with eight PEs were measured to confirm its performance. It consumes down to 8 pJ/cycle. Compared to existing commercial solutions, this is at least 10× less in terms of overall energy and 2–3× faster.

REFERENCES

[1] V. Raghunathan, C. Schurgers, S. Park, and M. B. Srivastava, "Energy-aware wireless microsensor networks," *IEEE Signal Process. Mag.*, vol. 19, no. 2, pp. 40–50, Mar. 2002.

[2] C. Walravens and W. Dehaene, "Design of a low-energy data processing architecture for wsn nodes," in *Proc. Design, Automat. Test Eur. Conf. Exhibit.*, Mar. 2012, pp. 570–573.

[3] H. Karl and A. Willig, *Protocols and Architectures for Wireless Sensor Networks*, 1st ed. New York: Wiley, 2005.

[4] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*, 4th ed. San Mateo, CA: Morgan Kaufmann, 2007.

[5] S. Mysore, B. Agrawal, F. T. Chong, and T. Sherwood, "Exploring the processor and ISA design for wireless sensor network applications," in *Proc. 21th Int. Conf. Very-Large-Scale Integr. (VLSI) Design*, 2008, pp. 59–64.

[6] J. Backus, "Can programming be liberated from the von neumann style?" in *Proc. ACM Turing Award Lect.*, 1977, pp. 1–29.

[7] L. Nazhandali, M. Minuth, and T. Austin, "SenseBench: Toward an accurate evaluation of sensor network processors," in *Proc. IEEE Workload Characterizat. Symp.*, Oct. 2005, pp. 197–203.

[8] P. Sanders and J. Träff, "Parallel prefix (scan) algorithms for MPI," in *Proc. Recent Adv. Parallel Virtual Mach. Message Pass. Interf.*, 2006, pp. 49–57.

[9] G. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.

[10] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Reading, MA, USA, Addison Wesley, 2010.

[11] G. E. Blelloch, "Prefix sums and their applications," Carnegie Mellon Univ., Pittsburgh, PA: USA, Tech. Rep. CMU-CS-90, Nov. 1990.

[12] M. Hempstead, J. M. Lyons, D. Brooks, and G.-Y. Wei, "Survey of hardware systems for wireless sensor networks," *J. Low Power Electron.*, vol. 4, no. 1, pp. 11–29, 2008.

[13] V. N. Ekanayake, C. Kelly, and R. Manohar "SNAP/LE: An ultra-low-power processor for sensor networks," *ACM SIGOPS Operat. Syst. Rev. - ASPLOS*, vol. 38, no. 5, pp. 27–38, Dec. 2004.

[14] V. N. Ekanayake, C. Kelly, and R. Manohar, "BitSNAP: Dynamic significance compression for a lowenergy sensor network asynchronous processor," in *Proc. IEEE 11th Int. Symp. Asynchronous Circuits Syst.*, Mar. 2005, pp. 144–154.

[15] M. Hempstead, D. Brooks, and G. Wei, "An accelerator-based wireless sensor network processor in 130 nm cmos," *J. Emerg. Select. Topics Circuits Syst.*, vol. 1, no. 2, pp. 193–202, 2011.

[16] B. A. Warneke and K. S. J. Pister, "An ultra-low energy micro-controller for smart dust wireless sensor networks," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*. Feb. 2004, pp. 316–317.

[17] M. Hempstead, M. Welsh, and D. Brooks, "Tinybench: The case for a standardized benchmark suite for TinyOS based wireless sensor network devices ," in *Proc. IEEE 29th Local Comput. Netw. Conf.*, Nov. 2004, pp. 585–586.

[18] O. Girard. (2010). "OpenMSP430 processor core, available at opencores.org," [Online]. Available: http://opencores.org/project,openmsp430

[19] H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. 100, no. 2, pp. 153–161, Feb. 1971.