International Journal of Applied Sciences, Engineering and Management
ISSN 2320 – 3439, Vol. 10, No. 01, January 2021, pp. 01-07

Study on DSP architectures for Wireless Sensor Nodes

T. R. S. Chandran
Guest Lecturer, Krishna University college of Engineering and Technology, Machilipatnam, Andhra Pradesh, India.
Email id: ramji.subhash2@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. But maintain an appropriate processing elements and algorithms the power and architecture can be organized and Sensor networks can manage efficiently. In this paper we concentrated on the comparison of design of Sensor nodes by using BK and KS adders. An efficient architecture is proposed for developing a signal processing block for wireless sensor networks**.**

Keywords: WSN, Folded Tree, DSP, Wireless Sensor

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 ultralow- energy WSN digital signal processor by further exploiting this and other characteristics unique to WSNs.

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 bottleneck [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.

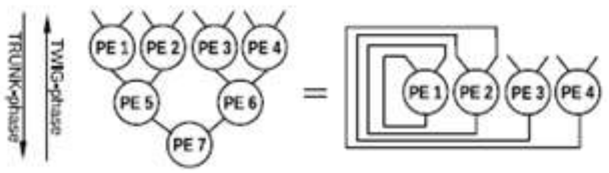


Fig1: A binary tree (left, 7 PEs) is functionally equivalent to the no folded tree topology (right, 4 PEs) used in this architecture.

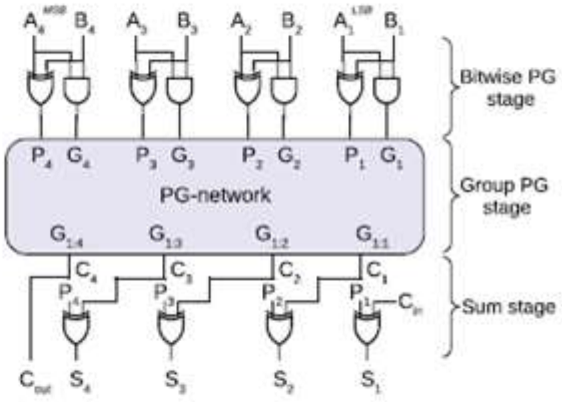


Fig 2: Addition with Propagate-Generate (PG) logic.

III. Existing 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 ($Pi = Ai \oplus Bi$ and $Gi = Ai \cdot Bi$) are fed as (Pi , Gi)-pairs to the group PG logic stage, which implements the following expression: $(Pi , Gi) \circ (Pi+1, Gi+1) = (Pi \cdot Pi+1, Gi + Pi \cdot Gi+1)$. It can be shown this-operator has an identify element $I = (1, 0)$ and is associative.

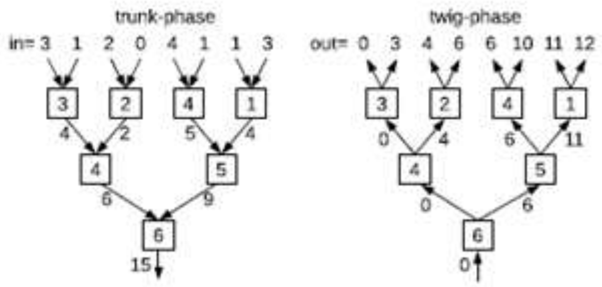


Fig 3: Example of prefix calculation with sum –operator using blelloch's generic approach in a trunk and twing phase

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* mod 2*)*. The rest is calculated as follows:

input = [2, 3, 8, 7, 6, 2, 1, 5]
keep = [0, 1, 0, 1, 0, 0, 1, 1]
prefix = [0, 1, 1, 2, 2, 2, 3, 4]
result = [3, 7, 1, 5].

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.

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 log2*(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 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. 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. 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 trunkphase 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.

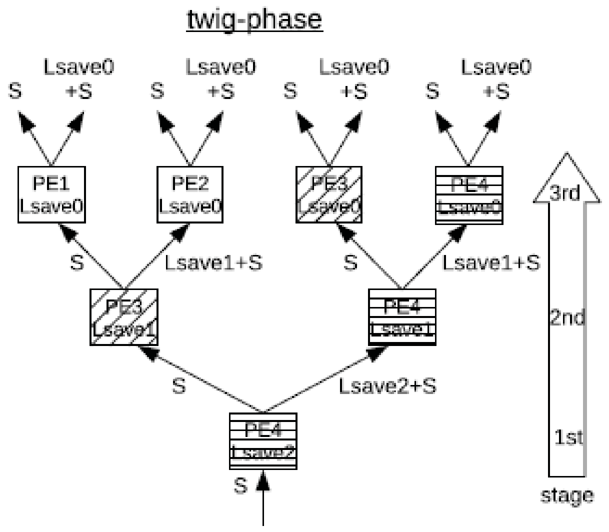


Fig 4: folded tree

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 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.

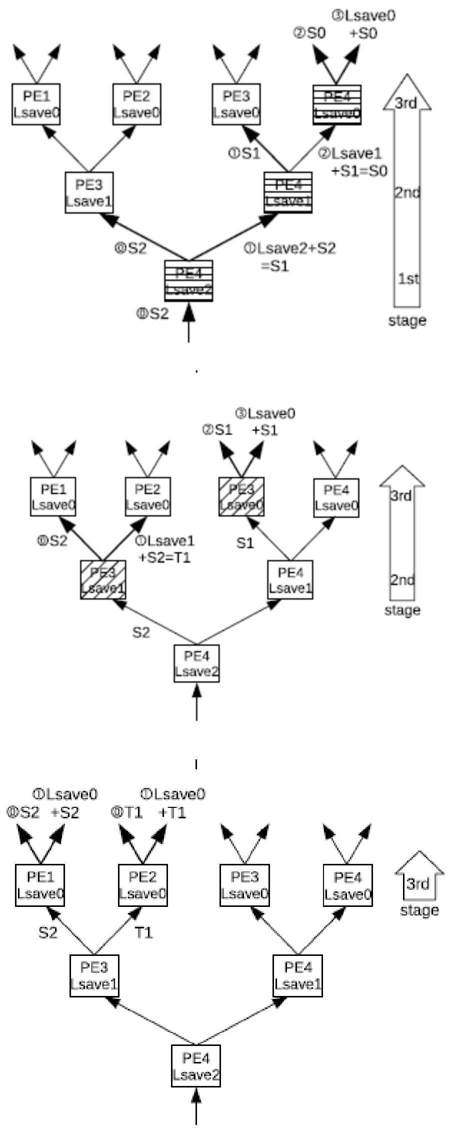


Fig. 5. Activity of different PEs during the stages of the twig-phase for a 4-PE folded tree: PE4 (top), PE3 (center), PE1 and PE2 (bottom).

V. PROPOSED WORK

In this paper the wireless sensor network is constructed and designed using parallel prefix adders mostly KS and BK adders. A comparison is formulated for these different architectures which are unique with their performance and low power capabilities. Using these architectures, the folding and unfolding are applied at the trans-receiver side. Individually each technique act as encryption and decryption

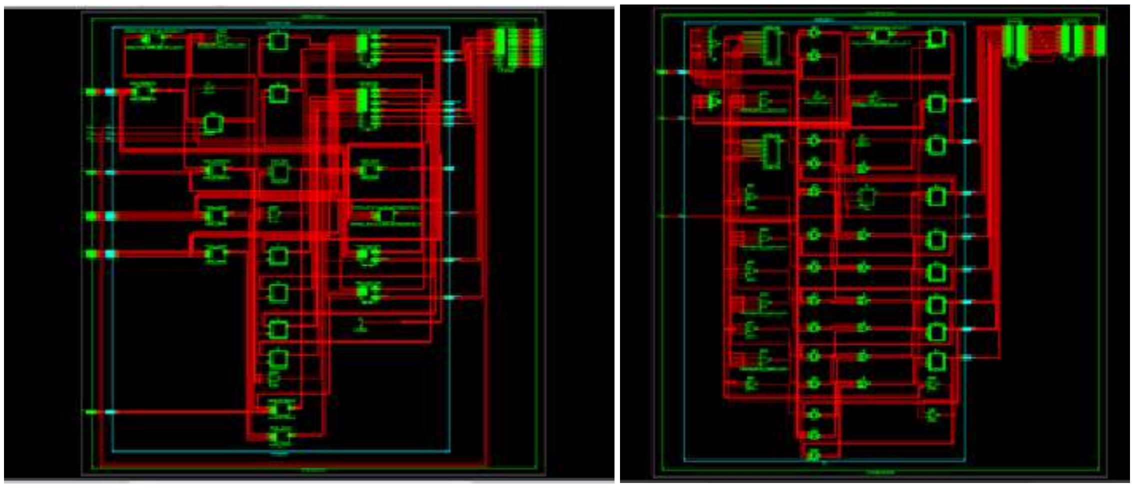
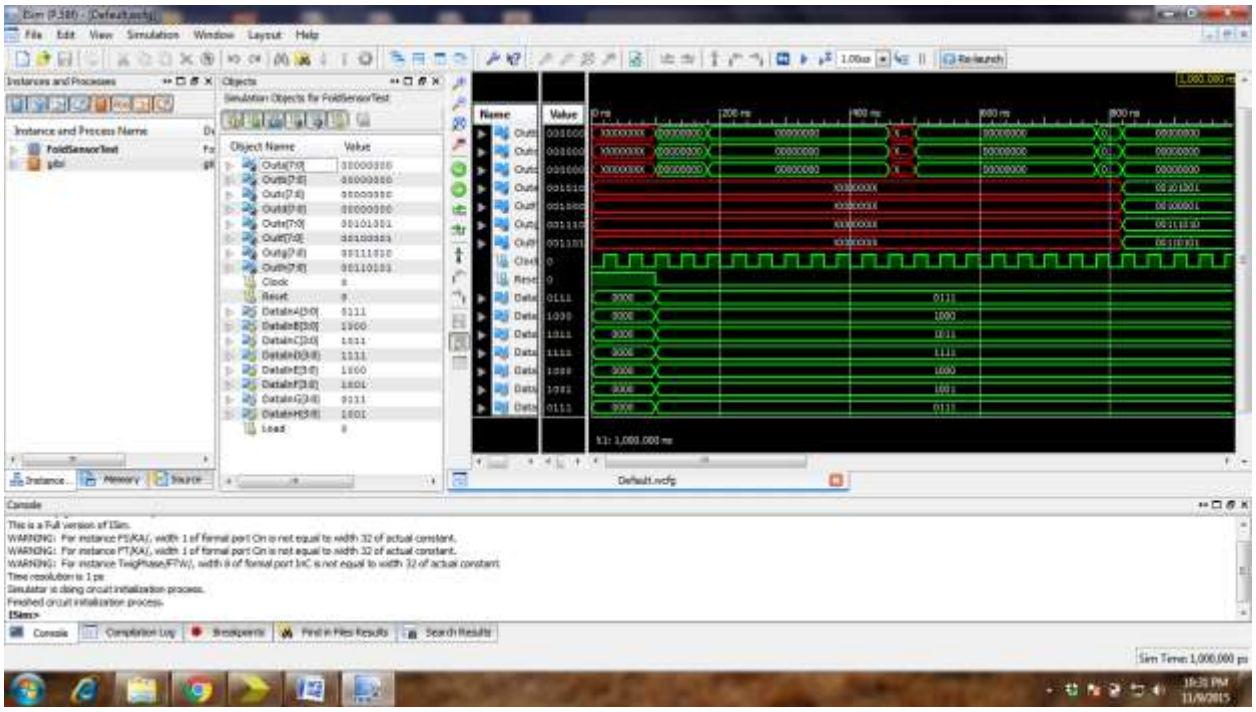


Fig 6: RTL schematic for fold & Unfold of WSN using BK



Simulation Objects for FoldSensorTest

Object Name	Value
Outa[7:0]	00000000
Outb[7:0]	00000000
Outc[7:0]	00000000
Outd[7:0]	00000000
Oute[7:0]	00101001
Outf[7:0]	00100001
Outg[7:0]	00111010
Outh[7:0]	00110101
Clock	0
Reset	0
DataInA[3:0]	0111
DataInB[3:0]	1000
DataInC[3:0]	1011
DataInD[3:0]	1111
DataInE[3:0]	1000
DataInF[3:0]	1001
DataInG[3:0]	0111
DataInH[3:0]	1001
Load	0

```
This is a Full version of ISim.
WARNING: For instance FS/KA/, width 1 of formal port Cin is not equal to width 32 of actual constant.
WARNING: For instance FT/KA/, width 1 of formal port Cin is not equal to width 32 of actual constant.
WARNING: For instance TwigPhase/FTW/, width 8 of formal port InC is not equal to width 32 of actual constant.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
```

Fig 7: Simulation result of Folding of WSN using BK adder

Comparison table for BK and KS

Criteria	BK	KS
No LUT for Folding	196	176
No. LUT for Unfolding	152	147
Power	73mW	73mW

Conclusion

A DSP architecture is designed and implemented using parallel prefix adders by BK and KS adders. Using this a folding and unfolding technique is applied for the data between the nodes of the sensors networks. A comparison table is drawn between the BK and KS adders where number of LUT and power are compared and by deducing KS based architecture is considered be area efficient and BK is high performance base for Sensor network systems. The architecture is considered as according to the applications of the Sensor networking system.

References

[1] V. Raghunathan, C. Schurgers, S. Park, and M. B. Srivastava, “Energyaware 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-lowpower 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 microcontroller 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.