An Embedded Programmable Processor for Compressive Sensing Applications

Mehdi Safarpour, Ilkka Hautala, Olli Silvén
Center for Machine Vision and Signal Analysis
University of Oulu, Oulu, Finland
mehdi.safarpour [AT] oulu.fi

Abstract- An application specific programmable processor is designed based on the analysis of a set of greedy recovery Compressive Sensing (CS) algorithms. The solution is flexible and customizable for a wide range of problem dimensions, as well as algorithms. The versatility of the approach is demonstrated by implementing Orthogonal Matching Pursuits, Approximate Messaging Passing and Normalized Iterative Hard Thresholding algorithms, all using a high-level language. Transported Triggered Architecture (TTA) framework is employed for the efficient implementation of macro operations shared by the algorithms. The performance of the CS algorithms on ARM Cortex-A15 and NIOS II processors has also been investigated, and empirical comparisons are presented. The flexible hardware design implemented on an FPGA achieves up to 7.80Ksample/s recovery at a power dissipation of 42μJ/sample and beats both ARM and NIOS in total power consumption.

Keywords—Compressive Sensing; Signal Reconstruction; Embedded Processor; IoT.

I. INTRODUCTION

The Compressive Sensing (CS) theory has paved the way for efficient sparse signal techniques and instruments [1]. Based on the CS theory, the number of measurements needed for exact signal reconstruction depends on the sparsity degree rather than bandwidth. Analog to Information Convertors (AIC) [2] are examples of devices that exploit signal sparsity for reduced sampling rates in the context of the CS framework [3]. A major issue with the compressive sensing framework is the computational complexity of CS reconstruction. The recovery algorithms introduced so far have been shown to possess polynomial computational complexity [4] that can be a particular problem with embedded applications.

A common assumption in CS based systems is that the recovery takes place in a remote cloud server, or a node in the sensor network with an unlimited supply of energy [5]. Unfortunately, this is not the case in many applications where the signal must be at least partially reconstructed locally for further decisions and adaptions. To tackle this challenge, application specific ASIC and FPGA based hardware designs have been proposed for CS recovery algorithms [6]–[8]. These hardware accelerator implementations usually reach satisfactory performance, but lack flexibility, which is a problem if further design iterations become necessary. Changing the reconstruction algorithm, updating it, or even modifying the reconstruction parameters, may require a major hardware redesign. Furthermore, the single purpose designs do not respond to other tasks required in an IoT node.

These limitations have been partially answered, for example in [6], [9] and [7] which provide controls for changing some design parameters, such as the problem dimension, algorithm constraints, etc. However, these are lacking compared to the flexibility provided by software implementations. In addition, a sensor node is supposed to carry out other tasks as well, which require extra resources. Software implementations on general purpose processors, though providing excellent flexibility, being multi-purpose, lack efficiency and speed in comparison with specific hardware designs [10]. Concerning application specific programmable designs, it has been observed that the practical CS signal reconstruction algorithms share a great deal of common macro operations including sorting, min-max, and common algebraic operations, while each algorithm has been designed on the basis of a different set of assumption.

In this paper, we utilize the above observation and propose an application specific CS reconstruction processor that is optimized for a variety of recovery algorithms. We also provide an empirical comparison of different algorithms on selected embedded platforms.

The motivation for this investigation stems from the opportunity to replace conventional signal compression in some applications with CS based measurements [5]. In other words, the signal is captured by an ADC, and CS sampling is carried out for the sampled and quantized signal for compression. This has been shown to be a more efficient compression method (it only comprises one matrix-vector multiplication) in sensor nodes in some cases. The design presented in [11] is an example of a processor for this class of applications. Also in our design, a pseudo random number generator is included to support these purposes.

The main contribution of this paper is a programmable, application specific embedded processor designed for sparse signal recovery. It enables pure software implementations of CS algorithms, hence supporting cost-effective software reconfigurability. We demonstrate the flexibility of the design by implementing different sparse recovery algorithms, and compare the resulting power consumptions and recovery rates to two other embedded processors

II. BACKGROUND

A. Compressive Sensing

Sampling theory requires the (average) sampling rate to be at least as high as twice of the maximum frequency of the signal bandwidth ($f_s > 2 \times f_{max}$), without considering the information

content of the signal [1]. On the contrary, compressive sensing theory leverages signal sparsity (low information content) into reduced sampling rates and provides a framework for acquisition and recovery of sparse signals with a significantly reduced number of required samples. Based on CS theory, a length-N signal like $X \in \mathbb{R}^{N\times 1}$, with a sparsity degree of k, i.e. with only k significant values in the sampling or transform domain, requires only M measurements ($M << N$) to ensure exact recovery, with overwhelming probability.

In compressive sensing, the measurements are obtained through multiplying an $M \times N$ matrix (Φ matrix or measurement matrix) in the signal vector X, which results in an incomplete set as follows:

$$Y_{M\times 1} = \Phi_{M\times N} X_{N\times 1} \quad (1)$$

where, $Y \in R^{M\times 1}$ is the measurements vector (Fig. 1).

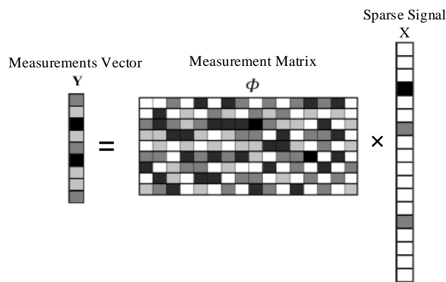

Fig. 1. Measurement of the sparse signal

Since ($M << N$), the above is an underdetermined system, so an infinite number of solutions exists that satisfy (1). However, when the sparsity assumption is introduced, a solution $\hat{X}$ can be recovered through solving the following l_1 norm minimization problem [1]:

$$\widehat{X'} = \arg\min \|\widehat{X'}\|_1 \ \text{ subject to } \ Y = \Phi \widehat{X'} \quad (2)$$

This recovery process is illustrated in Fig. 2. Please note that it is not necessary for the original signal X to be sparse in the sampling domain. The CS sampling framework still applies if there exists a sparse representation of X in a transform domain, i.e., $X = \Psi^{-1}a$, where a is the sparse transformed representation of X and Ψ is the transform matrix. An essential condition in CS is low incoherence between sampling matrix Φ, and the transform matrix Ψ. In such a case, Eq. (2) should be modified to the following to solve for a instead of X.

$$\hat{a} = \arg\min \|\hat{a}\| 1 \text{ subject to } \ Y = \Phi\Psi^{-1}\hat{a} \quad (3)$$

Once $\hat{a}$ is known, $\hat{X}$ is recovered through $\widehat{X} = \Psi^{-1}\,\hat{a}$. Incoherence of Φ and Ψ matrices is necessary, which requires Φ matrix to be random in practice [4].

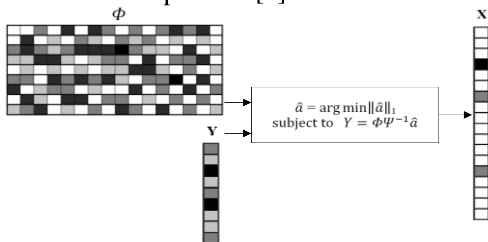

Fig. 2. CS based recovery process

The computational challenge in CS signal recovery is in solving an $l1$-norm minimization problem. This was originally tackled through convex optimization methods. However, a family of greedy algorithms has been introduced that are much less demanding in terms of computational complexity and memory requirement. Our CS signal recovery processor design targets these algorithms.

B. Recovery Algorithms

In this section, we briefly outline the CS reconstruction algorithms of interest to give a background for further discussion on the design choices. We only consider greedy algorithms due to their practicality in our considered application domain.

We implemented three popular greedy algorithms: Orthogonal Matching Pursuit (OMP) [12] , Approximate Messaging Passing (AMP) [13], and Normalized Iterative Hard Thresholding (N-IHT) [14]. All implementations were written in C programming language. The algorithms were run on three embedded processors: Altera NIOS II-f core, ARM cortex-1,5 and our proposed Transport-Triggered Architecture (TTA) [15], [16] template based design. It should be noticed that the first and the last processors were implemented on an FPGA, while the ARM was included in a system chip.

OMP is a greedy algorithm introduced as an extension to the well-established Matching Pursuit algorithm [17]. The OMP algorithm iteratively finds the best matrix columns that correspond to the non-zero coefficients of the sparse signal, and then performs a least squares (LS) optimization in the subspace formed from current and previously selected columns.

AMP and the IHT algorithms do not require LS in each iteration, and instead perform simple vector truncation, which results in an iterative completion of the sparse signal. The parameters, such as step size and threshold, are critical in the performance of AMP and IHT algorithms. The optimum parameters of AMP are chosen based on the experimental results of [13] . In the case of the IHT algorithm, a different flavor of algorithm called Normalized-IHT was implemented, where the step size is automatically determined in each iteration.

The recovery algorithms share a great deal of common operations, which we exploit in our design. Our aim has been to have an attractive combination of performance, energy efficiency, and design flexibility. This should allow for making algorithmic and parameter changes after the deployment of the design.

In this work, to solve overdetermined systems of equations, the QR decomposition (QRD) [18] method was employed for the OMP algorithm. AMP and NIHT algorithms are basically different from OMP in the sense that these algorithms refine the residue and estimate the signal in a greedy thresholding manner.

III. IMPLEMENTATION

The TTA application specific processor framework has a low programmability overhead, and generally achieves a close to ASIC performance [19]. TTA is somewhat similar to Very Long Instruction Word (VLIW) architecture, but its bypass network is exposed to the programmer. In comparison to VLIW, it enjoys better scalability, i.e. design complexity grows linearly with the increasing number of functional units and register files.

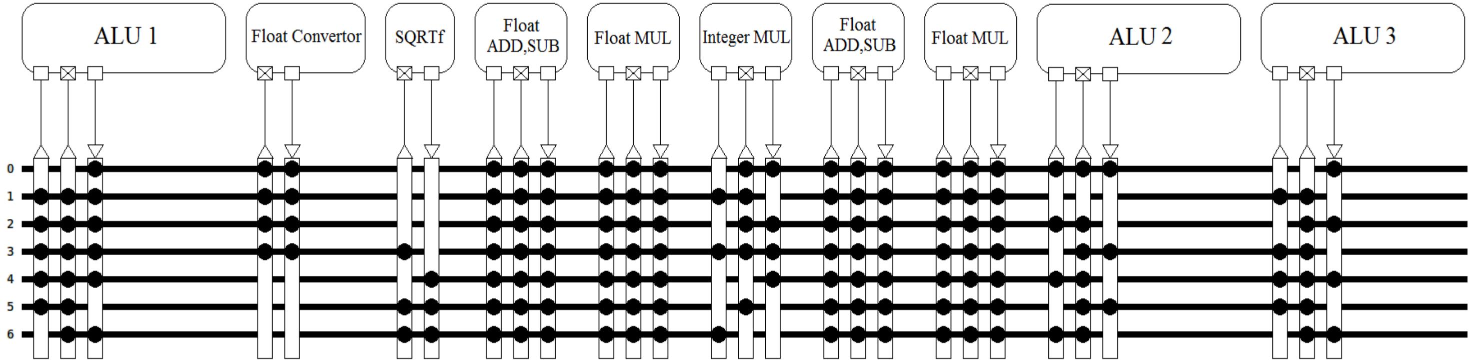

Fig. 3. A partial view of implemented processor architecture in the design software

The operations (instructions) are defined as data transports through common data paths (buses) between Function Units (FU) and Register Files (RF). The execution of an operation happens as a side effect of data transportation to the FUs. When data is written to the input registers of an FU, the FU is triggered, and after defined latency, the result is available in the output register of the FU, and can be directly moved to an input register of the same or another FU. These direct moves help to avoid the register files from becoming hotspots. Energy is also saved, because power-hungry multi-ported register files may not be needed at all.

In this work, the TCE toolchain (TTA-based Co-design Environment [20]) was employed for design and optimization of the processor. The processor was later implemented and tested for verification on an FPGA.

The number of connections to the FUs in the TTA architecture was optimized based on profiling the target algorithms to reduce the instruction word length. This also leads to lower power consumption of the instruction memory and the interconnections. As our goal was to optimize the processor for a variety of CS reconstruction algorithms that share many macro operations, we focused on optimizing for those macro operations rather any specific algorithm.

Using the TCE toolchain, the optimization comprised of counting the number of times each functional unit is triggered. Starting from a basic TTA processor with integer and floating-point function units, the algorithms of interest were profiled, identifying hotspots. For instance, the substantial number of integer operations from address calculations in matrix operations indicated respective optimization potential and guided both the processor and software design.

CS measurement and recovery requires the generating of a random matrix. To address this need, a pseudo-random number generator FU, using Linear Feedback Shift Register, was designed and added to the base processor. The list of function units in the designed TTA is given in Table 1.

Table 1. Details of designed processor

FUNCTIONAL UNIT	QUANTITY
FLOAT- MULTIPLIER	4
FLOAT- DIVIDER	1
INTEGER-MULTIPLIER	4
INTEGER UNIT	4
FLOAT SQRT	1
RANDOM GENERATOR	1
SUBTRACTOR	4
INSTRUCION MEMORY	76.7KB
DATA MEMORY	1 (256 KB)

The designed processor is partly depicted in Fig. 3. Based on analysis, floating point multiplications were found to be the most common micro operation in each considered algorithm. On the other hand, the multiplications are rarely simple scalar multiplications, rather they are often vector by matrix or matrix by matrix operations. Consequently, performance can be boosted by vectorizing the code and respective support at the hardware level. Special care was taken in designing the matrix multiplication function to exploit instruction level parallelism as far as possible.

IV. RESULTS AND DISCUSSION

The designed TTA processor and NIOS II-f core [21] were implemented on a Cyclone IV-EP4CE115F29C7 FPGA. The NIOS II-f is a rather advanced 32-bit RISC pipelined processor, equipped with dynamic branch prediction and instruction, and data memory caches. The detailed FPGA synthesis results are presented in the following table for NIOS II and the designed TTA processor. Moreover, an ODROID XU3 board [22] was used to perform the same CS algorithms on the ARM Cortex-15 processor. The system chip is provided with power consumption sensors that can be employed to read power dissipation at processor cores, DRAM and GPU.

Table 2. FPGA synthesis report for Cyclone IV-EP4CE115F29C7

	TTA	NIOS II-f (Customized)
Total Logic Elements	23,505 (21%)	9,495 (8%)
Total Registers	7057	2914
Total Memory bits	2,726,357 (68%)	2,931,328 (74%)
DSP Blocks	20	11
Max. Clock	62.92 MHz	79.61 MHz

for signal window size (max) = 256

Since the base NIOS II is an integer processor, it is very slow for our application (2,20s for OMP). Therefore, a custom floating-point instructions unit (including floating-point division) was added to the processor [23]. A great deal of logic elements is consumed for the floating-point functional units. In TTA, we have four floating point multipliers, whereas in NIOS II, we only use one.

For consistency and fair comparison, window sizes of 256, with 25% of Nyquist rate sampling were used in every test. The power consumptions for the TTA and NIOS processors were measured in vivo through measuring the board power difference (dynamic power), and through gate level simulation in Model Sim and Quartus software. In the case of the ARM

processor, power consumption was measured through querying the internal sensors.

Fig. 4 presents the times needed by the recovery algorithms to achieve given Signal to Noise ratios (SNR) on the designed TTA processor. Since the OMP algorithm converges earlier that the other two algorithms for highly sparse signals, less sparse signals were used for this experiment. As we can observe, if exact recovery is not the goal, but lower SNR results can be used, the reconstruction rate can be increased.

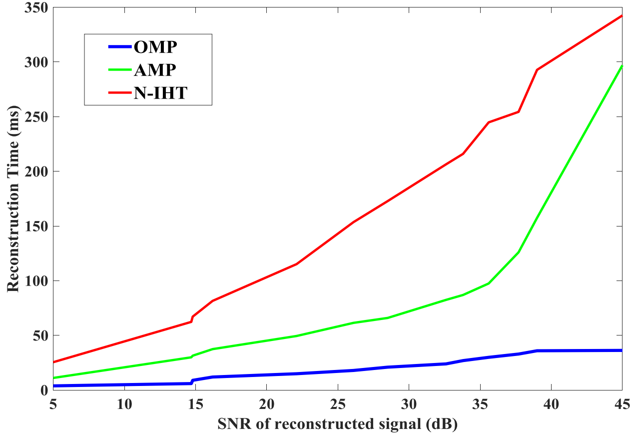

Fig. 4. Comparison of required time for given reconstruction quality (8% signal occupation)

Our measured estimated and simulated dynamic power consumptions almost match. Consequently, we hypothesize that even the simulated static power consumption is close to the actual figure. Unfortunately, we were unable to measure it with our experimentation board.

Table 3 summarizes the results for all algorithms and platforms. The data and instruction memory power consumptions in our proposed design are 18mW and 43mW, respectively.

Table 3. Performance for different algorithms and platforms

		SNR	NIOS II-f Customized	Proposed TTA	ARM Cortex-A15
Reconstruction Time [a]	OMP	70dB	260ms	33ms	1.45ms
	AMP	70dB	2.5s	350ms	5.0ms
	N-IHT	50dB	6.2s	730ms	23.8ms
Power Consumption	Dynamic	Mes.	195 mW	174 mW	1.7W
		Sim.	220 mW	194 mW	-
	Static		99 mW	101 mW	N/A
	Total [b]		351 mW	327 mW	2.7W
Clock Frequency			50MHz	50MHz	2GHz
Energy per Sample (Best)			391 µJ/S	42 µJ/S	15 µJ/S
Implementation Platform			FPGA	FPGA	Chip
Approximate GEs (Core)			~39807	~95656	-
Total Memory (Instruction + Data)			350 KBs	337KB	2 GB

[a] 2% signal occupation, 256 samples and 64 CS measurements
[b] 30mW for I/O is reported for all FPGA implementations

The static power is almost constant since the used FPGA and tools lack support of power gating of unused blocks. Note that the NIOS and TTA processors are both on FPGA platforms, while the ARM is included in a system chip. The energy per recovered sample was lowest for the ARM in our experiments. However, assuming an implementation using 28nm CMOS technology, we can speculate the energy consumption of the proposed TTA to be around 130mW at a 1.2GHz clock rate. This would result in 1.3ms reconstruction time for the OMP algorithm, and 630nJ/s energy per sample. The energy efficiency is almost 20 times that of the ARM-A15 core. This speculation is based on the results for the processor implemented in [24], with similar architecture, and roughly the same number of gate elements for each core as in our proposed design.

CS recovery algorithms are sequential by nature but include steps that can be fully parallelized. Depending on implementation, it might be necessary to transfer data between different GPU kernels, which will introduce a data transfer overhead. Although, the reported results of GPU implementations are promising for long signals [25], in an embedded setting, we are generally looking for small length signal recovery. The reported FPGA implementations in the literature show superior efficiency over ours; for example, reference [26] reports a recovery time of 23.27µs (at 110MHz) on a Virtex-5 FPGA with 530mW dynamic power consumption for reconstruction of a signal with the same number of measurements and length as ours, but, as discussed earlier, these implementations are single purpose. To the best knowledge of authors, no application specific programmable processor is reported for CS recovery applications.

For fair comparison of the algorithms, one must take into consideration the window length of the signal, the sparsity degree of the test signal, the hyperparameters of the algorithms, such as termination criteria and desired reconstruction quality. From Fig. 4, we observed that the OMP algorithm is fastest in reconstruction, whereas the AMP and IHT algorithms that are known to be computationally cheaper, appear to be slower. This is due to the low sparsity degree and short signal length. The OMP algorithm gives better performance for less sparse signals [27], and here the experiments were done with signals with less than 10% occupancy. The IHT algorithm's performance is relatively independent from the sparsity degree, and the performance of AMP is less sensitive to sparsity degree than the OMP [27], [28]. These issues are rather strong practical arguments for flexible designs.

Although CS specific implementations [6], [7], [9] provide high efficiency, they suffer from a low degree of flexibility and customization, which hinders exploring the design space for a variety of parameters for each specific algorithm. For the highly varying applications of IoT nodes, single purpose hardware for each task is not always affordable, but flexibility is preferred without sacrificed performance and energy efficiency.

V. SUMMARY

The proposed TTA based CS recovery processor demonstrates satisfactory Energy per Sample performance already on an FPGA platform. As a fully programmable design, it can be utilized for other tasks, too. All three well-known CS algorithms were written completely in a high-level language, showing the versatility of the toolchain used. Based on the

results, and considering embedded systems design constraints, the OMP algorithm seems to be the best option for short length signal recovery in embedded settings.

VI. ACKNOWLEDGMENTS

The support of the Academy of Finland for the ICONICAL project is gratefully acknowledged.

REFERENCES

[1] E. J. Candès and M. B. Wakin, "An introduction to compressive sampling," *IEEE Signal Process. Mag.*, vol. 25, no. 2, pp. 21–30, 2008.

[2] S. Kirolos *et al.*, "Analog-to-information conversion via random demodulation," in *Design, Applications, Integration and Software, 2006 IEEE Dallas/CAS Workshop on*, 2006, pp. 71–74.

[3] M. Safarpour, R. Inanlou, M. Charmi, O. Shoaei, and O. Silvén, "ADC-Assisted Random Sampler Architecture for Efficient Sparse Signal Acquisition," *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, 2018.

[4] D. Needell and J. A. Tropp, "CoSaMP: Iterative signal recovery from incomplete and inaccurate samples," *Appl. Comput. Harmon. Anal.*, vol. 26, no. 3, pp. 301–321, 2009.

[5] F. Chen, A. P. Chandrakasan, and V. M. Stojanovic, "Design and analysis of a hardware-efficient compressed sensing architecture for data compression in wireless sensors," *IEEE J. Solid-State Circuits*, vol. 47, no. 3, pp. 744–756, 2012.

[6] P. Maechler *et al.*, "VLSI design of approximate message passing for signal restoration and compressive sensing," *IEEE J. Emerg. Sel. Top. Circuits Syst.*, vol. 2, no. 3, pp. 579–590, 2012.

[7] F. Ren and D. Markovic, "A Configurable 12-237 kS/s 12.8 mW Sparse-Approximation Engine for Mobile Data Aggregation of Compressively Sampled Physiological Signals.," *J Solid-State Circuits*, vol. 51, no. 1, pp. 68–78, 2016.

[8] M. Mayer, N. Görtz, and J. Kaitovic, "RFID tag acquisition via compressed sensing," in *RFID Technology and Applications Conference (RFID-TA), 2014 IEEE*, 2014, pp. 26–31.

[9] Z. P. Ang and A. Kumar, "Real-time and low power embedded ℓ 1-optimization solver design," in *Field-Programmable Technology (FPT), 2013 International Conference on*, 2013, pp. 168–175.

[10] J. D. Blanchard and J. Tanner, "Performance comparisons of greedy algorithms in compressed sensing," *Numer. Linear Algebra Appl.*, vol. 22, no. 2, pp. 254–282, 2015.

[11] J. Constantin *et al.*, "TamaRISC-CS: An ultra-low-power application-specific processor for compressed sensing," in *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, 2012, pp. 159–164.

[12] J. A. Tropp and A. C. Gilbert, "Signal recovery from random measurements via orthogonal matching pursuit," *IEEE Trans. Inf. Theory*, vol. 53, no. 12, pp. 4655–4666, 2007.

[13] C. A. Metzler, A. Maleki, and R. G. Baraniuk, "From denoising to compressed sensing," *IEEE Trans. Inf. Theory*, vol. 62, no. 9, pp. 5117–5144, 2016.

[14] A. Kyrillidis and V. Cevher, "Recipes on hard thresholding methods," in *Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2011 4th IEEE International Workshop on*, 2011, pp. 353–356.

[15] H. Corporaal, "Microprocessor Architectures: from VLIW to TTA," 1997.

[16] J. Boutellier, O. Silven, and M. Raulet, "Automatic synthesis of TTA processor networks from RVC-CAL dataflow programs," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, 2011, pp. 25–30.

[17] S. Mallat and Z. Zhang, "Matching pursuit with time-frequency dictionaries," Courant Institute of Mathematical Sciences New York United States, 1993.

[18] M. Karkooti, J. R. Cavallaro, and C. Dick, "FPGA implementation of matrix inversion using QRD-RLS algorithm," in *Asilomar Conference on Signals, Systems, and Computers*, 2005.

[19] P. Jääskeläinen, V. Guzma, A. Cilio, T. Pitkänen, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Multimedia on Mobile Devices 2007*, 2007, vol. 6507, p. 65070X.

[20] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, "HW/SW Co-design Toolset for Customization of Exposed Datapath Processors," in *Computing Platforms for Software-Defined Radio*, Springer, 2017, pp. 147–164.

[21] P. P. Chu, *Embedded SOPC design with NIOS II processor and VHDL examples*. John Wiley & Sons, 2011.

[22] J. Ivković, A. Veljović, B. Ranđelović, and V. Veljović, "ODROID-XU4 as a desktop PC and microcontroller development boards alternative," in *Proc. 6th Int. Conf.(TIO)*, 2016.

[23] D. Etiemble, S. Bouaziz, and L. Lacassagne, "Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing," in *Embedded Systems for Real-Time Multimedia, 2005. 3rd Workshop on*, 2005, pp. 61–66.

[24] I. Hautala, J. Boutellier, and O. Silven, "Programmable 28nm coprocessor for HEVC/H. 265 in-loop filters," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, 2016, pp. 1570–1573.

[25] M. Andrecut, "Fast GPU implementation of sparse signal recovery from random projections," *ArXiv Prepr. ArXiv08091833*, 2008.

[26] Ö. Polat and S. K. Kayhan, "High-speed FPGA implementation of orthogonal matching pursuit for compressive sensing signal reconstruction," *Comput. Electr. Eng.*, vol. 71, pp. 173–190, 2018.

[27] I. L. M. Gutierrez, H. A. Fuentes, and K. Winbladh, "Implementation of imaging compressive sensing algorithms on mobile handset devices," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2012 Seventh International Conference on*, 2012, pp. 252–259.

[28] J. A. Tropp, "Greed is good: Algorithmic results for sparse approximation," *IEEE Trans. Inf. Theory*, vol. 50, no. 10, pp. 2231–2242, 2004.