Low-Power ASIP Architecture Exploration and Optimization for Reed-Solomon Processing

Andreas Genser[1], Christian Bachmann[1], Christian Steger[1], Jos Hulzink[2], Mladen Berekovic[3]
[1]*Institute for Technical Informatics, Graz University of Technology, Austria*
[2]*IMEC NL, Holst Centre Eindhoven, The Netherlands*
[3]*Technical University Braunschweig, Germany*
{*andreas.genser, christian.bachmann, steger*}*@tugraz.at*
jos.hulzink@imec-nl.nl
berekovic@ida.ing.tu-bs.de

Abstract

The advent of the mobile age has heavily changed the requirements of today's communication devices. Data transmission over interference-prone wireless channels requires additional steps of data processing, such as forward error correction, to ensure reliable communication. In this work we present RS(63,55) Reed-Solomon encoding and decoding algorithms according to the IEEE 802.15.4a standard [2] executed on dedicated application-specific processor architectures. Algorithmic as well as architectural modifications to speed up execution and well-known low-power techniques to reduce the power consumption are discussed. The speed-up for our proposed designs compared to a general purpose baseline architecture is up to two orders of magnitude. Power reduction due to clock-gating and guarded evaluation results in a 40% power drop and the energy consumption is decreased up to 60x.

1. Introduction

Modern technologies require data processing and communication within very tight energy constraints. The Interuniversity Microelectronics Centre (IMEC) proposed the Human++ program [1] to evaluate human-related data, such as ECG, EMG or EEG by means of wireless sensor nodes. Data transmission is carried out by ultra-wide band (UWB) communication based on the IEEE 802.15.4a standard [2]. Noisy communication channels can cause transmission errors, which require forward error correction (FEC) techniques to guarantee data integrity. Reed-Solomon (RS) codes have been widely used as a FEC technique to enhance the reliability of data communication systems. Adding additional information to the data to be transmitted on the sender side enables the receiver to perform error correction (see Fig. 1). However, RS algorithms require additional computations, which increase the energy consumption of the overall system. The limited energy budget for mobile domain applications and their steadily increasing complexity offer little room for RS computations. Hence, energy-efficient RS execution becomes increasingly essential.

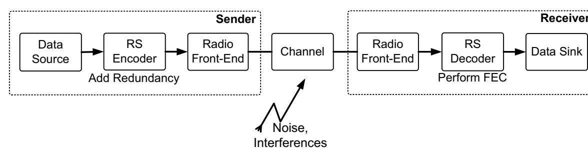

Figure 1. Reed-Solomon forward error correction principle

Previous RS works range from ASICs [3] to more flexible processor implementations, both commercial ones as presented in [4] as well as in academia [5], [6]. ASIC implementations are beneficial in terms of hardware complexity and power consumption, but are very limited in flexibility. Processor implementations on the other hand, are very flexible but lack energy efficiency. An application-specific instruction-set processor (ASIP) combines both, energy-efficiency and flexibility. This paves the way for low-power and flexible RS implementations required for wireless and low-power applications as defined in the IEEE 802.15.4a standard. We apply optimizations on multiple layers of abstraction, ranging from algorithm improvements, processor architecture modifications down to gate-level optimizations, such as clock-gating and guarded evaluation. Optimizations are carried out on scalar, vector and very long instruction word (VLIW) ASIPs to determine the most apt processor architecture for RS algorithms. In contrast to many other RS implementations we focus on both, the optimization of RS encoding and decoding. Moreover, this work aims at low power and energy efficiency rather than throughput.

The remainder of this paper is structured as follows. Section 2 provides a brief insight in RS encoding and decoding algorithms. In Section 3 an ASIP design flow as well as target architectures are presented. Optimizations applied to these architectures are discussed in Section 4, Section 5 presents performance, power, energy and area results and conclusions are drawn in Section 6.

2. Reed-Solomon Encoding and Decoding Algorithm

2.1. Reed-Solomon Encoding

Reed-Solomon codes are specified by $RS(n,k)$. A number of k symbols, each m-bit wide can be denoted by $m(x)$. They are extended with $2t = n-k$ parity symbols, which leads to an n-symbol code word $c(x)$ (see Fig. 2).

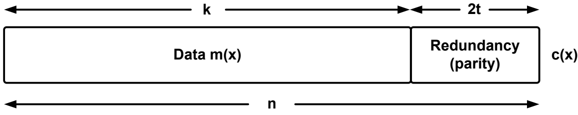

Figure 2. RS code word structure: data word $m(x)$ of length k extended with $2t$ parity symbols

Each element of the polynomials $m(x)$ and $c(x)$ is based on the extended finite field $GF(2^m)$. $g(x)$ is a generator polynomial of degree $2t$. Given these polynomials, the RS encoding equation can be expressed as

$$c(x) = m(x)X^{2t} + m(x)\ mod\ g(x) \tag{1}$$

where $c(x)$ represents the code word to be transmitted, which is a concatenation of the data word $m(x)$ and the parity symbols generated by $m(x)\ mod\ g(x)$.

The error correction capability of an $RS(n,k)$ algorithm can be given as $t = \frac{n-k}{2}$. Referring to $RS(63,55)$ defined in IEEE 802.15.4a [2] and used in this work, up to four errors are correctable.

2.2. Reed-Solomon Decoding

Reed-Solomon decoding is composed of four major blocks as depicted in Fig. 3.

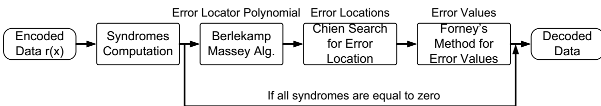

Figure 3. RS decoding block scheme

The received code word $r(x)$ of length n can be given as

$$r(x) = c(x) + e(x) \tag{2}$$

Errors introduced by a noisy channel or other sources of interference are given as $e(x)$. The syndromes computation block determines whether errors have occurred during data transmission according to (3). If all syndrome values S_i are equal to zero, $r(x)$ is error-free and the remaining decoding blocks can be skipped.

$$S_i = r(x)\ |_{X=\alpha^i} \quad for\ 1 \leq i \leq 2t \tag{3}$$

If $r(x)$ is error-prone the Berlekamp-Massey algorithm determines the error locator polynomial $\Lambda(x) = 1 + \Lambda_1\ x + \Lambda_2\ x^2 + ... + \Lambda_\nu\ x^\nu$. It contains information about the locations of errors within the received code word $r(x)$. Once $\Lambda(x)$ has been found, its roots are analyzed by the Chien-search algorithm. The Chien-search algorithm tests whether $\Lambda(\alpha^i)$ evaluates to zero for $0 \leq i < n$. If $\Lambda(\alpha^i) = 0$, i indicates an error position in $r(x)$. These roots mark the actual error locations in $r(x)$. Finally, the Forney algorithm computes error values based on the error magnitude polynomial $\Omega(x)$ required to correct errors in $r(x)$ as illustrated in (4).

$$e_i(x) = \frac{\Omega(x)}{x\ \Lambda'(x)}\ |_{X=\alpha^i} = \frac{\Omega(\alpha^i)}{\alpha^i\ \Lambda'(\alpha^i)} \tag{4}$$

$\Omega(x)$ is defined as

$$\Omega(x) = \Lambda(x)\ [1 + S(x)] \tag{5}$$

where $S(x) = S_1\ x + S_2\ x^2 + ... + S_{2t}\ x^{2t}$.

3. ASIP Design

Various vendors provide tools tailored to ASIP design [7], [8], [9]. The designer can take advantage of features provided in these tools to speed up the design process.

3.1. Target Design Flow

This work is based on the IP designer tool-suite provided by *Target Compiler Technologies* [7]. The tool-flow, including source and architecture description languages as well as software development and HDL generation tools, is depicted in Fig. 4. Processor architectures are described in nML, a highly structured language to specify hardware at a high-level of abstraction. A retargetable compiler supports the generation of machine code for arbitrary application-specific hardware architectures. Machine code can be executed by a cycle-accurate instruction-set simulator (ISS) to investigate how application and architectural changes impact on performance. Once the application meets performance requirements, HDL code can be generated.

3.2. Architectures

The processor used as a starting point in our work is a pipelined 16-bit scalar Harvard RISC architecture as depicted on the left-hand side of Fig. 5. It is composed of a 16-bit arithmetic logical unit (ALU), a register file of 16 16-bit registers, load-store unit, branch unit and instruction decoder. Also denoted in Fig. 5 is an extension unit, containing RS specific instruction-set extensions to speed up RS algorithm execution.

The right-hand side of Fig. 5 shows our second architecture, a vector processor. Its scalar functional units are similar

to those of the scalar processor architecture. Furthermore, single instruction multiple data (SIMD) instructions are employed enabling ALU operations to be executed in parallel. To perform RS operations concurrently, the instruction-set is extended with RS specific vector instructions. The number of vector elements has been defined in accordance to the $RS(63, 55)$ standard. Hence, a vector processor architecture with the capability to execute eight operations in parallel has been chosen.

The third considered architecture is a VLIW processor, which consists of two scalar and two vector units (see Fig. 6). Instruction-level parallelism enables two scalar and two SIMD instructions to be executed concurrently. To speed up RS execution, finite field extensions are included in both, the scalar and vector units.

4. ASIP Optimizations

4.1. Custom Instructions

RS encoding and decoding algorithms are heavily based on finite field operations. It is trivial to carry out a finite field addition in hardware, which is a simple bitwise XOR instruction. This instruction is incorporated in the standard instruction-set of the considered processor architectures. In contrast, finite field multiplication is more complicated. Two approaches have been investigated in this work, (i) look-up table (LUT) based and (ii) hardware finite field multiplication.

LUT based finite field multiplication requires two pre-computed tables, *log_table* and *Alog_table*.

- $log_table[\alpha^i]$ returns i
- $Alog_table[x]$ returns α^x

The multiplication of two finite field operands α^i and α^j can be given as

$$\alpha^i \cdot \alpha^j = \alpha^{(i+j)\ mod\ (m-1)}. \quad (6)$$

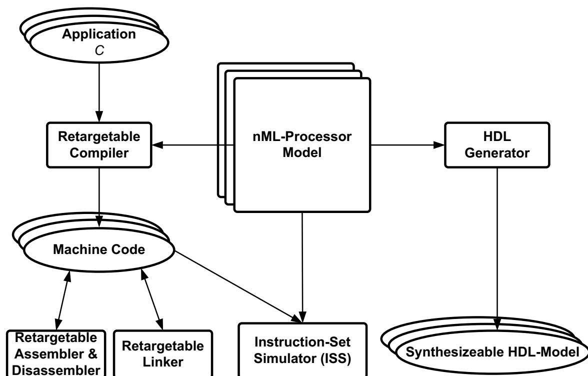

Figure 4. ASIP design flow as provided by *Target Compiler Technologies* [7]

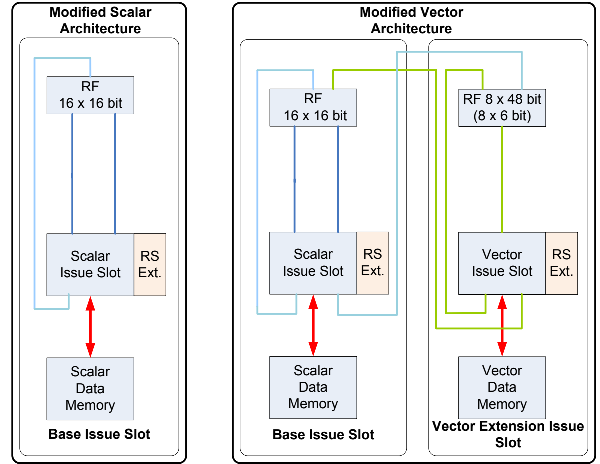

Figure 5. Scalar and vector processor architecture comprising RS specific extensions

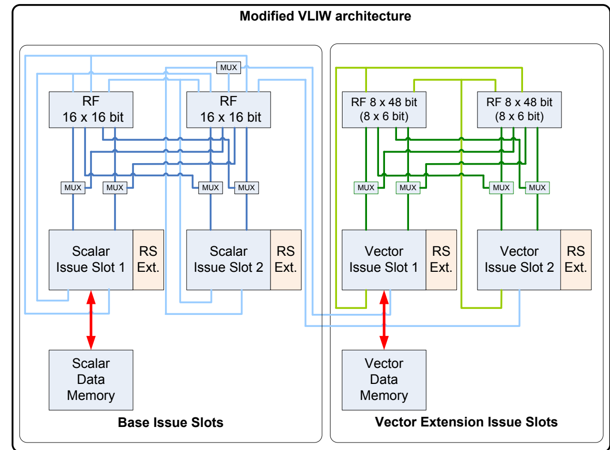

Figure 6. VLIW processor architecture comprising RS specific extensions

A possible implementation of a finite field multiplication is illustrated in Listing 1.

```
if((alpha_i != 0) && (alpha_j != 0))
{
  result = log_table[alpha_i] +
           log_table[alpha_j];
        if(result >= m-1)
          result -= m-1;
        return Alog_table[result];
}
return 0;
```

Listing 1. LUT-based software implementation of finite field multiplication

No additional hardware is required for this operation but the LUTs occupy extra memory. In addition, three table look-ups, one addition and a conditional statement have to be executed. The main shortcomings of this approach are extra memory accesses as well as multi-cycle execution. This

Table 1. Overview of implemented custom instructions, 8 scalar elements per vector

Instr.	Input params.	Output params.
Finite field mult.	(scl, scl)	scl
Vector finite field mult.	(vec, vec)	vec
Vector finite field summation	(vec)	scl
Vector finite field mult.-acc.	(vec, vec)	scl
Vector custom shift left/right	(vec, scl)	vec
Vector set custom element	(vec, scl, scl)	vec
Vector get custom element	(vec, scl)	scl

brings down algorithm performance significantly.

Finite field multiplication performed in hardware can be achieved by a finite field multiplier proposed by Mastrovito in [10]. It solely consists of combinational logic and provides a result within one clock cycle without memory accesses. A low-complexity scalar design approach for a Mastrovito multiplier given in [11] has been implemented in this work. To allow for vector finite field multiplication, scalar finite field multipliers are duplicated. Besides finite field custom instructions, vector manipulation instructions are implemented and provided as intrinsic functions. They are summarized in Tab. 1.

4.2. Algorithmic Optimizations

Scalar RS encoding and decoding algorithms are implemented on the standard scalar processor architecture. For the improved version of the scalar RS algorithms LUT-based finite field multiplications are replaced by custom instructions, which are provided as intrinsic functions.

Algorithm vectorization is carried out by restructuring algorithms to utilize the vector units as much as possible and accelerate the algorithms by employing vector custom instructions (see Tab. 1). Except the Berlekamp-Massey algorithm, which incorporates much control flow (e.g. jump instructions etc.), RS encoding and decoding algorithms can be well vectorized.

Multiple scalar and vector units are available in the VLIW processor architecture, which enables parallel execution of two scalar and two vector instructions. RS algorithms executed on the VLIW processor architecture are similar to those executed on the vector processor architecture. Potential performance gains are achieved by the compiler that detects instructions without data dependencies and schedules them to maximize parallel execution.

4.3. Low-Power Techniques

Low-power techniques introduced in this work are (i) clock-gating and (ii) guarded evaluation. Clock-gating logic is inserted across the processor to prevent clocking parts, which currently do not change their state. Power savings in the range of 30 - 40% can be achieved [12]. In addition, guarded evaluation determines and prevents redundant combinational switching logic by inserting additional logic [13]. This brings power savings of up to 30%. Both techniques are supported by Target's IP designer tool-suite.

5. Experimental Results

This chapter discusses results obtained by carrying out performance and power measurements. For RS encoding simulations an arbitrary data word $m(x)$ is used. On the RS decoder side one randomly generated error e_i at a random position i is introduced in the code word $c(x)$ during simulation.

5.1. Performance Results

Performance evaluation has been performed by means of the instruction-set simulator (ISS) of Target's IP designer tool-suite. The number of executed cycles has been evaluated for each implementation. Moreover, our RS algorithms were ported to a cycle-accurate simulator of the Texas Instruments TMS320C64x platform [14] and are compared to an alternative RS implementation on a Starcore SC140 [4]. Moreover we compare performance to a recent embedded vector processor (EVP) RS decoder implementation published in [6]. Unfortunately, the ASIP implementation published in [5] does not contain any results. Note that cycle counts in [4] and [6] are published for $RS(255, 239)$ implementations, hence they are downscaled for a fair comparison to our work.

Fig. 7 depicts cycle count measurements for RS encoding. Migrating from the standard scalar processor architecture to its modified counterpart, which makes use of hardware finite field multiplication instead of a LUT based approach accounts to a cycle count reduction of 7x. Vectorization improves the performance by another 8x. Finally the VLIW

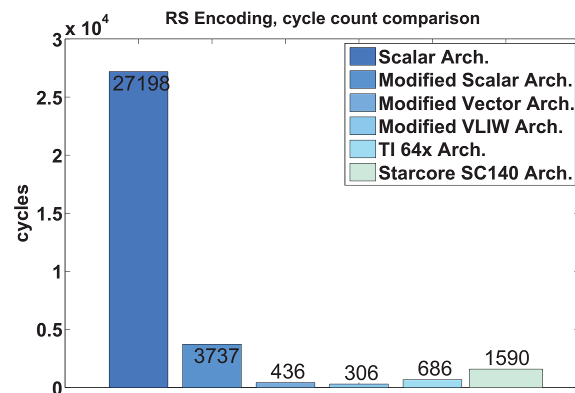

Figure 7. Cycle counts of different processor architectures for RS encoding

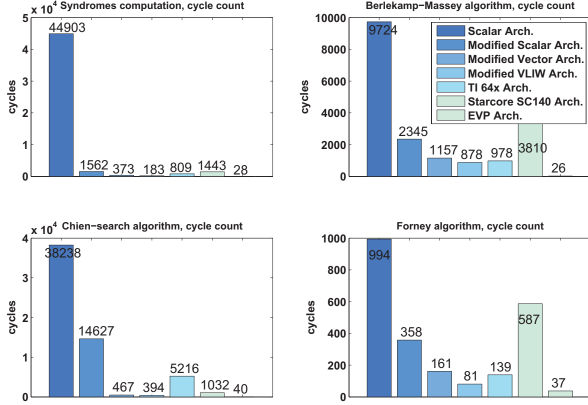

Figure 8. Cycle counts of different processor architectures for RS decoding

processor architecture brings down the RS encoding algorithm to 306 cycles. This implies an overall improvement of 88x. The comparison to the TI TMS320C64x and the Starcore SC140 illustrates that both, the vector and VLIW processor architecture, are superior.

In Fig. 8 the cycle count measurements for all RS decoding blocks can be found. The syndromes computation could be improved by 245x, the Berlekamp-Massey algorithm by 11x, the Chien-search algorithm by 97x and the Forney algorithm by 12x compared to the general purpose baseline architecture. The vector and VLIW processor deliver again higher performance compared to the TI TMS320C64x and the Starcore SC140. The EVP benefits from excessive vector processing (32 vector elements) and therefore outperforms all compared implementations.

Especially the RS encoding algorithm as well as the syndromes computation and Chien-search algorithm benefit heavily from vector custom instructions. The Berlekamp-Massey and Forney algorithm are inherently more serially structured, which makes them more difficult to vectorize. Compared to alternative implementations the highest performance is delivered by the EVP.

5.2. Power Results

To obtain power values, all processors have been synthesized to 100MHz using *Cadence Encounter*. The underlying technology is a 90nm low-power High-Vt library. Power values discussed in the following have been obtained by *Primetime PX* provided by *Synopsys*. Tab. 2 gives a power consumption comparison of all RS algorithm implementations for all processor architectures. The scalar processor accounts to a power consumption of 1mW on average, whereas the modified VLIW processor marks the upper range with 3mW. Unfortunately, we could not find any meaningful power values for the TI TMS320C64x nor for the Starcore SC140. The EVP $RS(255, 239)$ decoder implementation dissipates 1mW/MHz based on a 90nm process [6].

Table 2. Power consumption comparison of different processor architectures

Processor Average Power Consumption (mW)				
	Scalar	Scalar Ext.	Vec. Ext.	VLIW Ext.
RS Enc.	0.93	1.23	1.45	2.97
RS Dec. (Syndr.)	0.93	1.13	1.8	3.87
RS Dec. (BM)	0.91	0.96	1.07	2.28
RS Dec. (Chien)	0.9	1.04	1.05	2.14
RS Dec. (Forney)	0.87	0.94	1.8	3.87
RS Dec.	0.91	1.14	1.44	3.19

Table 3. Low-power techniques applied to the vector processor architecture for the RS encoding algorithm

Avg. Power Consumption (mW), low-power options, *RS* Enc.			
	Vec. Ext.	Vec. Ext., CG	Vec. Ext., CG&GE
Power (mW)	2.46	1.89	1.45
Improvement (%)		23.2	41.1

The impact of applied low-power techniques is listed in Tab. 3 by means of the vector processor architecture. The modified vector processor architecture without implemented low-power features consumes 2.46mW on average while RS encoding. Inserting clock-gating (CG) logic brings down power dissipation to 1.89mW, which is a decrease of 23.2%. Additionally added guarded evaluation (GE) logic leads to 1.45mW power consumption on average. Hence, the overall power consumption reduction achieved is 41.1%. The power consumption of the scalar as well as the VLIW processor architecture scales down similarly to the vector processor architecture when applying low power measures.

5.3. Energy Results

Fig. 9 illustrates the energy consumption for RS encoding of one data word $m(x)$. It is shown that the amount of energy consumed during algorithm execution can be drastically reduced by migrating from the standard scalar architecture to its modified version incorporating RS extensions. The vector and VLIW processor architectures are even more energy-efficient. The achieved energy reduction is a factor of 40x. Similar results can be given for RS decoding. An energy efficiency gain of 4x to 62x could be achieved compared to the original general purpose baseline architecture. We additionally compare our energy results to the EVP published in [6]. The lower cycle count of the EVP as discussed in Section 5.1 does not compensate the EVP's higher power consumption in terms of energy.

Energy measurements emphasize that modifications made to the processor architectures are beneficial. The slightly higher power consumption of the vector and VLIW processor architectures can be compensated in terms of energy,

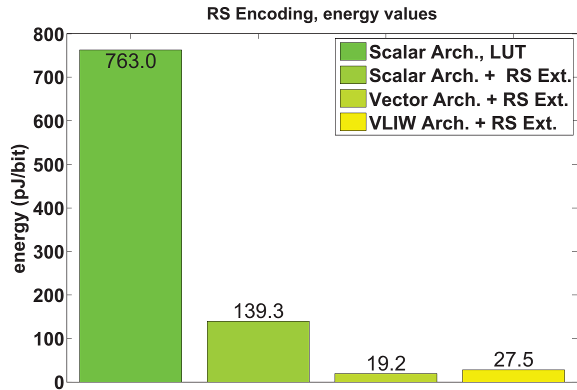

Figure 9. Energy consumption of different processor architectures for RS encoding

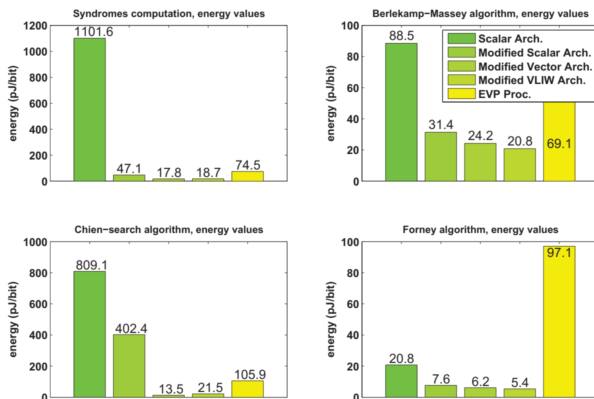

Figure 10. Energy consumption of different processor architectures for RS decoding

due to significantly better algorithm run-times. Additionally, the vector and VLIW processor architecture consume less energy compared to the EVP.

5.4. Processor Area

The overall area of the processors including memories accounts to $0.064mm^2$ (22.7k gates) for the scalar, $0.13mm^2$ (46k gates) for the vector and $0.35mm^2$ (124k gates) for the VLIW processor architecture. The EVP in [6] consists of approx. 600k gates. Instruction-set extensions contribute up to approx. 5% of the total processor areas.

6. Conclusion

Low-power forward error correction has become increasingly important with the advent of mobile devices. In this paper we present scalar, vector and VLIW application-specific instruction-set processors comprising hardware extensions to implement low-power RS encoding and decoding algorithms. Alongside algorithm and processor architecture modifications, low-power techniques are introduced to bring down the processor's energy consumption. The migration from a general purpose baseline architecture to our application-specific scalar, vector and VLIW processor architectures accelerates RS algorithms by up to two orders of magnitude. Low-power techniques decrease the power consumption by 40%. The entirety of optimization techniques leads to significant improvements in run-time and energy efficiency while still maintaining greater flexibilty compared to an ASIC implementation.

Acknowledgments

We would like to thank *Target Compiler Technologies* for their support throughout this project.

References

[1] Human++: Emerging Technology for Body Area Networks, Gyselinckx, B. et al., IFIP, Nice, pp. 175–180, 2006.
[2] IEEE P802.15.4a Draft Amendment to IEEE Standard for Information technology - Telecommunications and information exchange between systems, New York, USA, 2007.
[3] High Throughput and Low Power Reed Solomon Decoder for Ultra Wide Band, Kumar, A. and Sawitzki, S., Intelligent Algorithms in Ambient and Biomedical Computing, Springer, part 3, pp. 299–316. 2004.
[4] Reed Solomon Encoder / Decoder on the StarCore SC140 / SC1400 Cores, Oz, J. and Naor, A., Freescale Semiconductor, App. Note AN2407, 2004.
[5] A new VLSI design for decoding of Reed-Solomon codes based on ASIP, Yuanxin, X. and Fang X. and Qingdong Y. and Peiliang Q. and Kuang W., ASIC, 448-451, 2001.
[6] Vectorization of Reed Solomon Decoding and Mapping on the EVP, Kumar, A. and van Berkel, K., DATE, pp. 450–455, 2008.
[7] Design of ASIPs in multi-processor SoCs using the Chess / Checkers retargetable tool suite, G. Goossens, D. Lanneer, W. Geurts, J. Van Praet, Int. Symp. on SoC, 2006.
[8] Silicon Hive Technology Primer, Whitepaper, SiliconHive.
[9] Rapid SOC Development Using Automatically Generated Processors, Whitepaper, Tensilica.
[10] VLSI Designs for Multiplication over Finite Fields GF (2^m), Mastrovito, E. D., AAECC-6, Springer-Verlag, London, UK, pp. 297–309, 1989.
[11] Systematic Design Approach of Mastrovito Multipliers over GF(2^m), Zhang, T. and Parhi, K. K., SIPS, pp. 507–516, 2000.
[12] Low Power Methodology Manual: For System-on-Chip Design, Keating, M. and Flynn, D. and Aitken, R. and Gibbons, A. and Shi, K., Springer, 2007.
[13] Guarded evaluation: pushing power management to logic synthesis / design, Tiwari, V. and Malik, S. and Ashar, P., ISPLED, pp. 221–226, 1995.
[14] Reed Solomon Decoder TMS320C64x Implem., Sankaran, J., Texas Instruments, 2000.