On The Design of A Dynamically Reconfigurable Function-Unit for Error Detection And Correction

Thilo Pionteck*[1], Thomas Stiefmeier*[2], Thorsten Staake*[3], and Manfred Glesner[4]

[1] University of Lübeck, Institute of Computer Engineering, 23538 Lübeck, Germany, pionteck@iti.uni-luebeck.de
[2] ETH Zürich, Wearable Computing Lab, CH-8092 Zürich, Switzerland, stiefmeier@ife.ee.ethz.ch
[3] University of St.Gallen, Institute of Technology Management, CH-9000 St. Gallen, Switzerland, thorsten.staake@unisg.ch
[4] Darmstadt University of Technology, Institute of Microelectronic Systems, 64283 Darmstadt, Germany, glesner@mes.tu-darmstadt.de

Abstract. This paper presents the design of a function-specific dynamically reconfigurable architecture for error detection and error correction. The function-unit is integrated in a pipelined 32 bit RISC processor and provides full hardware support for encoding and decoding of Reed-Solomon Codes with different code lengths as well as error detection methods like bit-parallel Cyclic Redundancy Check codes computation. The architecture is designed and optimized for the usage in the medium access control layer of mobile wireless communication systems and provides simultaneously hardware support for control-flow and data-flow oriented tasks.

1 Introduction

For wireless communication systems the capability of receivers to detect and correct transmission errors is of great importance. While error detection methods require bandwidth expensive retransmissions, error correction methods lead to a better bandwidth efficiency. Albeit this advantage, error correction codes are not often used for mobile wireless communication systems due to their decoding complexity. Software solutions would require powerful processors, leading to an unacceptable power consumption for battery powered mobile devices. Hardware solutions are often optimized for throughput, yet are inflexible and do not consider the requirements of mobile terminals. For mobile wireless communication terminals, the critical design parameter is not throughput but area efficiency and power consumption. Though works exist on area-efficient or (re)configurable Reed-Solomon decoders (e.g. [2, 15]), the potentials of dynamically reconfigurable approaches concerning hardware savings are often not

* The authors performed this work while at Darmstadt University of Technology

taken into consideration. However, temporal reuse of hardware within the decoding offers the best potential to achieve an area-efficient design. The inherent hardware overhead of reconfigurable solutions can be avoided by restricting the reconfiguration capabilities to one application area, resulting in a function-specific reconfigurable device.

In the following the design of a dynamically reconfigurable function-unit (RFU) supporting Cyclic Redundancy Checks (CRC) and Reed-Solomon Codes with different code lengths is presented. The RFU is optimized regarding area and fulfills the performance requirements for actual wireless communication standards. In combination with a processor, the RFU allows the design of a flexible solution for the MAC (*M*edium *A*ccess *C*ontrol) layer of WLANs. Reconfiguration can be done during runtime, allowing the processor to utilize all arithmetic components and memory elements of the RFU for additional tasks like multiplication in the Galois Field required for encryption standards like AES (*A*dvanced *E*ncryption *S*tandard) [1].

The outline of the contribution is as follows: Section 2 provides a short review of error detection and error correction codes. In Section 3, a reference design for a Reed-Solomon decoder is presented. This decoder was used as a starting point for the design of the function-specific reconfigurable architecture introduced in Section 4. Section 5 deals with the system integration of the RFU, and Section 6 presents performance and synthesis results. Finally, conclusions are given in Section 7.

2 Error Detection and Correction Codes

Error detection codes have efficiently been employed in many communication protocols. They enable the receiver to detect whether a received code word is corrupted or not. As the receiver does not have the information required for correcting the error, a retransmission of the corrupted data has to be initiated. Error correction codes extend the redundant part of the message with information so that errors up to a certain degree of corruption can be corrected. Thus the retransmission probability can be reduced considerably by using *forward error correction* (FEC).

2.1 Cyclic Redundancy Check

Cyclic Redundancy Check codes are a powerful subclass of error detection codes and are well-suited for detecting burst errors. The basic idea is to expand a k-bit message, described by a polynomial $u(x)$ with coefficients in {0,1}, with the remainder $R_{g(x)}$ of the division of $x^{n-k} \cdot u(x)$ by a $m = (n-k)$th-order generator polynomial $g(x)$ using modulo-2 arithmetic, resulting in an n-bit code word $v(x)$. If $v(x)$ is affected by an error polynomial $e(x)$, a receiver can check the integrity of the received data $v_e(x) = v(x) + e(x)$ by dividing $v_e(x)$ by $g(x)$. A non-zero remainder $r(x)$ indicates the presence of errors [14].

There exist two common approaches to perform CRC computation, a bit-serial and a bit-parallel one. The serial approach uses a linear feedback shift register (LFSR) based on the generator polynomial $g(x)$. One bit is processed per cycle which results in low performance. The parallel CRC computation method is based on multiplications in Galois Fields. Here, a message $u(x)$ is divided into blocks of length $m = n-k$ denoted by $W_i(x)$. Using the congruence properties of modulo-2 operations and the fact that the degree of $W_i(x)$ is less than m, the code word $v(x)$ can be written as $v(x) = W_{N-1} \otimes \beta_{N-1} \oplus \ldots \oplus W_0 \otimes \beta_0$, where $\otimes$ and $\oplus$ denote the multiplication and addition over a Galois Field $GF(2^m)$, respectively. The coefficients β_i depend only on the generator polynomial $g(x)$ and can be computed in advance. A more detailed description can be found in [11].

2.2 Reed-Solomon Codes

Reed-Solomon (RS) Codes are a very common group of systematic linear block codes and are based on operations in Galois Fields. A RS(n,k) code word $v(x)$ consists of n symbols of length m, divided into k message symbols and $(n-k)$ parity symbols. Up to $(n-k)$ symbol errors can be detected and $t = (n-k)/2$ symbol errors can be corrected.

The binary representation of the original data is segmented into k symbols of m bits. These symbols are interpreted as elements of a Galois Field $GF(2^m)$, constructed by a primitive polynomial $p(x)$ of degree m. The resulting message polynomial $u(x)$ is then multiplied by the polynomial x^{n-k} and added to the remainder polynomial $r(x)$ to form the code word polynomial $v(x) = x^{n-k} \cdot u(x) + r(x)$. The term $r(x)$ is the remainder of the division of $x^{n-k} \cdot u(x)$ by a generator polynomial $g(x)$ of degree $n-k$.

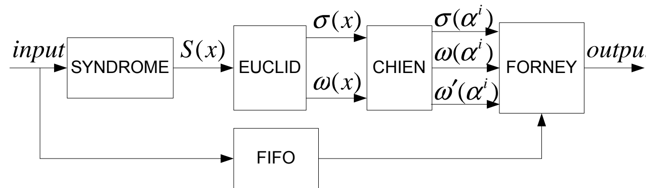


Fig. 1. Reed-Solomon Decoder Structure

Decoding Reed-Solomon Codes is much more complex. As shown in figure 1, the decoding process can be divided into four processing blocks.

Syndrome Calculation: First, the syndrome polynomial $S(x) = \sum_{i=0}^{2t-1} S_i \cdot x^i$ is determined. In case $S(x)$ is zero, the received word $w(x)$ can be assumed to be error free. S_i is defined as $S_i = \sum_{j=0}^{n-1} w_j \cdot \alpha^{ij}$, where α is a root of the primitive polynomial $p(x)$ using the power notation for elements in $GF(2^8)$.

Euclid's Algorithm: In the second step, the error locator polynomial $\sigma(x)$ and the error value polynomial $\omega(x)$ are calculated by solving the key equation $S(x) \cdot \sigma(x) = \omega(x) \; mod \; x^{2t}$. This is done by using Euclid's Algorithm which can be summarized as follows [9, 5]: Three temporary polynomials $R(x)$, $B(x)$ and $Q(x)$ are introduced. They are initialized as $R_{-1}(x) = x^{2t}$, $R_0(x) = S(x)$, $B_{-1}(x) = 0$ and $B_0(x) = 1$. For the i-th iteration, the equations $R_i(x) = R_{i-2}(x) - R_{i-1}(x) \cdot Q_{i-1}(x)$ and $B_i(x) = B_{i-2}(x) - B_{i-1}(x) \cdot Q_{i-1}(x)$ are solved, where $Q_{i-1}(x)$ is the quotient and $R_i(x)$ is the remainder of the divison of $R_{i-2}(x)$ by $R_{i-1}(x)$. This is done for s iterations until the degree of $R_i(x) < t$. The error locator polynomial is defined as $\sigma(x) = \frac{B_s(x)}{B_s(0)}$ and the error value polynomial as $\omega(x) = \frac{R_s(x)}{B_s(0)}$.
Chien Search: This block determines the error positions in the received symbol block. To this end, the roots α_i $(1 \leq i \leq 8)$ of the error locator polynomial $\sigma(x)$ are determined, e.g. it is checked whether $\sigma(\alpha^i) = 0$. This is done by means of an exhaustive search over all possible field elements α^i in $GF(2^8)$. In addition, the derivate $\sigma'(\alpha^i)$ is determined.
Forney Algorithm: The last step consists of calculating the error value $e_i = \frac{\omega(\alpha^i)}{\sigma'(\alpha^i)}$ $(0 \leq i \leq n-1)$. This value is added to the received symbol to correct the error.

3 Reed-Solomon Decoder Structure

In the following, the hardware design of a reference Reed-Solomon decoder is presented. The decoder is capable to support variable n and k values with an error correction capability of up to eight symbol errors ($t \leq 8$). This decoder structure is then mapped onto the function-specific reconfigurable architecture in Section 4.

3.1 Galois Field Arithmetic

All operations on Reed-Solomon Codes RS(n,k) are defined at byte-level, with bytes representing elements in a Galois Field $GF(2^8)$. Addition (and substraction) in $GF(2^8)$ result in a bitwise XOR operation denoted by $\oplus$. Multiplication and division are much more complex and depend on the used primitive polynomial. To date, numerous works have been devoted to the design of configurable Galois Field multipliers (e.g. see [8]). Most of them realize bit-serial architectures. Yet, for the proposed reconfigurable architecture, single cycle bit-parallel multipliers should be used in order to minimize the latency of the decoder.

The structure of a bit-parallel multiplier is presented in figure 2. Computation is done by a repeated application of a *xtime* (xt) operation, which is a left shift and a subsequent conditional bitwise XOR operation at byte-level [1, 10]. The structure of an xtime module for the primitive polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$ is shown in figure 3. In order not to restrict the

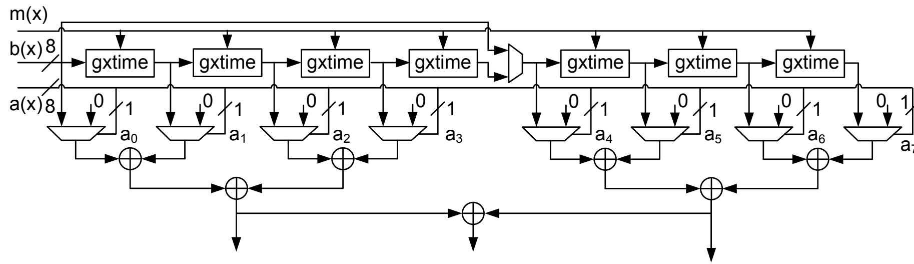


Fig. 2. Structure of a bit-parallel $GF(2^8)$ multiplier

design to a fixed primitive polynomial, an extended version of the *xtime* operation has been designed. This *gxtime* module can be configured to support any primitive polynomial. Its structure is shown in figure 4.

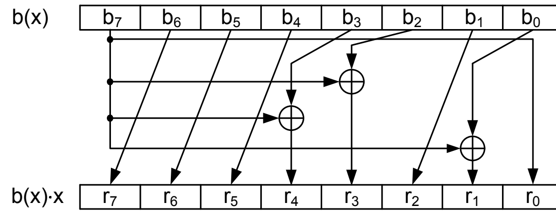


Fig. 3. *Xtime* Module

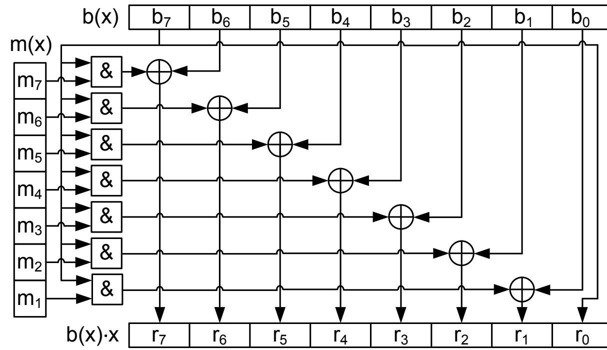


Fig. 4. *GXtime* Module

Division in Galois Fields is done by an inversion followed by a multiplication. As described in [4, 12] Euclid's Algorithm or Fermat's Theorem can be used

for the inversion. In the reference architecture a third method based on look-up tables is used. Although this approach requires more chip area than other methods, the look-up tables were chosen since they offer higher flexibility and higher speed. In the subsequent reconfigurable architecture the look-up tables can also be used for several other purposes.

3.2 Reed-Solomon Decoder Blocks

The hardware structure of the Reed-Solomon decoder is based on the block diagram presented in figure 1. For the *Syndrome Calculation*, a hardware design derived from [9, 15, 7] is used. This design mainly consists of $2t$ multiply-accumulate circuits and requires n clock cycles to calculate the syndrome polynomial.

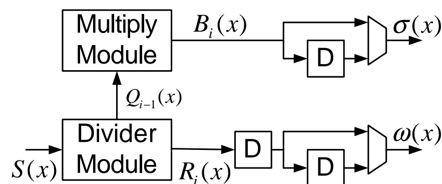


Fig. 5. Euclidean Algorithm Block [9]

Euclid's Algorithm is realized by using a structure as shown in figure 5. It is partitioned in a multiply and a divider module. The divider performs the division $\frac{R_{i-2}(x)}{R_{i-1}(x)}$ and generates the quotient $Q_{i-1}(x)$ and the new remainder $R_i(x)$ in each of the $\frac{n-k}{2}$ iterations. Each iteration requires three clock cycles. The multiply module uses $Q_{i-1}(x)$ to compute $B_i(x)$. The output of the multiply module forms the error locator polynomial $\sigma(x)$ while the result of the divider module is the error value polynomial $\omega(x)$. In total 25 $GF(2^8)$ multipliers are required: 16 for the divider module and 9 for the multiply module. A more detailed description of the realization of *Euclid's Algorithm* can be found in [9].

The *Chien Search* block requires two clock cycles for initialization and then computes $\omega(\alpha^i)$, $\sigma(\alpha^i)$ and $\sigma'(\alpha^i)$ simultaneously in each clock cycle. For the computation, 8 $GF(2^8)$ feed-back multipliers are used for determining $\omega(\alpha^i)$, and 10 for calculating $\sigma(\alpha^i)$. By calculating $\sigma(\alpha^i)$ as a sum of its coproducts $\sigma(\alpha^i) = \sigma_{even}(\alpha^i) + \sigma_{odd}(\alpha^i)$, the computation of $\sigma'(\alpha^i)$ does not require any extra hardware resources, as $\sigma_{odd}(\alpha^i) = \sigma'(\alpha^i) \cdot \alpha^i$.

The last block inside the Reed-Solomon decoder represents the *Forney Algorithm*. The division of $\frac{\omega(\alpha^i)}{\sigma'(\alpha^i)}$ is realized by an inversion and a multiplication. For inversion a look-up table is used. The error values e_i are added to the received symbols stored in a FIFO. In total n clock cycles are required. Figure 6 shows the error dectection combined with the error correction using *Forney Algorithm*.

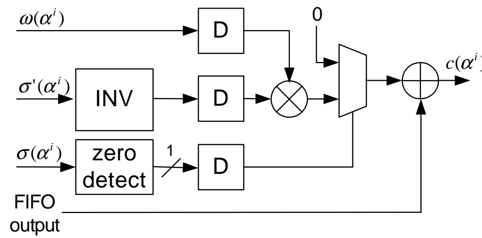


Fig. 6. Error Detection and Correction[9]

4 Reconfigurable Function-Unit

The design of the reconfigurable architecture was motivated by the idea to accomplish an integrated design capable to perform error detection and error correction algorithms as mentioned before. Designated to be part of a reconfigurable function-unit in a pipelined processor, the architecture allows direct access to all memory elements and arithmetic blocks, providing hardware support for additional tasks. The architecture is optimized in terms of hardware efficiency and flexibility, yet its flexibility is restricted to a degree which can be exploited by the dedicated application area. The RFU offers two levels of (re)configuration. On the one hand, the RFU can be configured to perform different tasks, e.g. CRC computation or Reed-Solomon Code encoding/decoding with different code lengths, on the other hand dynamic reconfiguration is used to achieve a huge hardware reuse within one task, resulting in an area-efficient design.

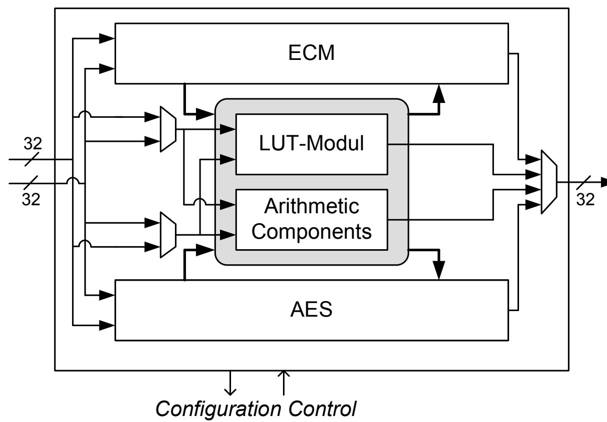


Fig. 7. Reconfigurable Function-Unit (RFU)

Figure 7 shows the structure of the RFU, which is composed of four different blocks. The *ECM* block (*E*rror *C*ontrol *M*odule) provides hardware support for error detection and error correction algorithms. The *AES* block is mentioned for the sake of completeness only. As the RFU is designated for the use in processors realizing the MAC-layer of WLANs, the *AES* block is integrated to provide hardware support for encryption/decryption tasks. A description of the *AES* block can be found in [13]. The two remaining blocks are the *LUT Module* and the *Common Resource* block. The *LUT Module* combines all memory elements of the *AES* and *ECM* blocks while the *Common Resource* block combines all complex arithmetic elements like the configurable Galois Field multipliers.

4.1 Error Control Module

The structure of the *ECM* block is depicted in figure 8. It comprises of two major blocks, *Block A* and *Block B*. The two blocks are derived from the symmetry of the underlying hardware structure. *Block A* is used for the *Syndrome Calculation* and RS encoding. *Block B* is used for *Forney Algorithm* and CRC encoding/decoding. *Euclid's Algorithm* and *Chien Search* require both, *Block A* and *Block B*. The structure of the cells inside *Block A* and *Block B* can be found in figure 9 and figure 10, respectively. Note, that the Galois Field multipliers shown in figure 9 and figure 10 are not realized in the cells but in the *Common Resource* block of the RFU.

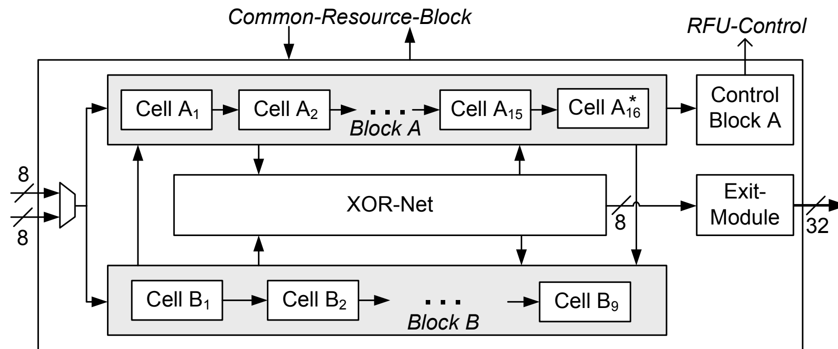


Fig. 8. Structure of Error Control Module

The *Input Module* of the *ECM* block stores constants required during processing and buffers the input data while the *Output Module* buffers the output data. These input and output buffers ease the programming of the RFU as no strict timing has to be met while accessing the *ECM* block. In addition, the gap between the 8-bit data path of the *ECM* and the 32-bit data path of the processor is bridged.

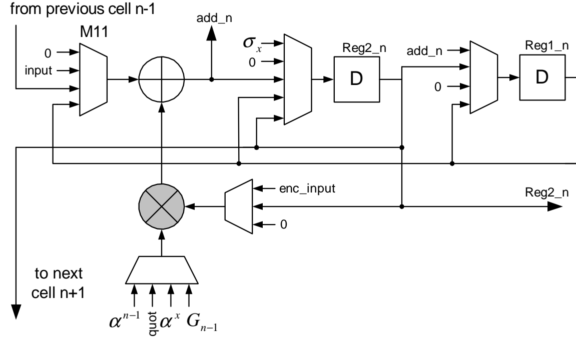


Fig. 9. Structure of Block A

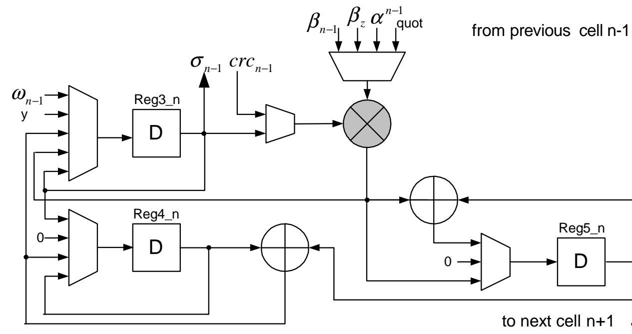


Fig. 10. Structure of Block B

4.2 Dynamic Reconfiguration

Computations like Reed-Solomon Code decoding require several reconfigurations at runtime. In order to release the processor from the control overhead of continuous reconfiguration, the control logic of the RFU is capable of performing a sequence of configuration steps autonomously. The structure of the reconfiguration control logic is shown in figure 11.

The main part of the control logic are its configuration tables, divided into three sub-tables (*Table 1*,*Table 2/4* and *Table 3/5*). These tables store the configuration vectors for the RFU and can autonomously be loaded by the control logic with configuration data from an external memory. The configuration tables are composed as follows: *Tables 3/5* are used for storing vectors which are fixed for a sequence of configurations. *Tables 2/4* store configuration vectors for different steps of a sequence of configurations and *table 1* determines the sequence of configurations. The execution of the sequence of configurations is controlled by a *run unit* inside the control logic.

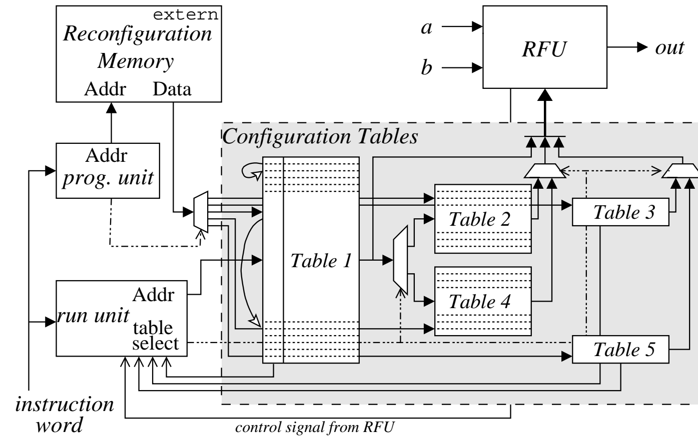


Fig. 11. Reconfiguration Control Logic

In each clock cycle a new entry in configuration *table 1* is selected by the *run unit*. In total 128 entries are provided. Processed configurations can be exchanged at runtime, enabling sequences of configurations with more than 128 steps. One table entry of *table 1* consists of eight bits. Three of these bits are either passed directly to the RFU or are used by the *run unit* for realizing (un)conditional jumps and loops. The other five bits of *table 1* are used as an address for *table 2* or *table 4*, consisting of 32 x 32 bits each. *Table 3* and *table 5* have a capacity of 32 bits each. Besides their usage for storing the fixed part of the configuration vector, they can also be used to provide the *run unit* with the number of iterations for repeated execution and with the offset values for (un)conditional jumps. This subdivision of the configuration memory allows to reduce the required size of configuration memory as not the complete configuration vector of 67 bits has to be stored for each step of the sequence. The division also eases the reprogramming. Only one table combination *table 2/3* or *table 4/5* can be active at runtime. The other combination can be reprogrammed without affecting the system.

5 Processor Integration

The RFU was integrated into a 32 bit 5 stage pipelined RISC core, derived from the DLX architecture [3]. Figure 12 shows the simplified datapath of the RISC processor with the integrated RFU. All changes made to the original datapath are highlighted. The RFU is placed next to the other function-units and utilizes the same datapaths to access the register files. The output of the RFU is non-registered. This constellation allows a full integration of the new function-unit in the pipeline structure of the processor. It also eases the integration into other processor designs.

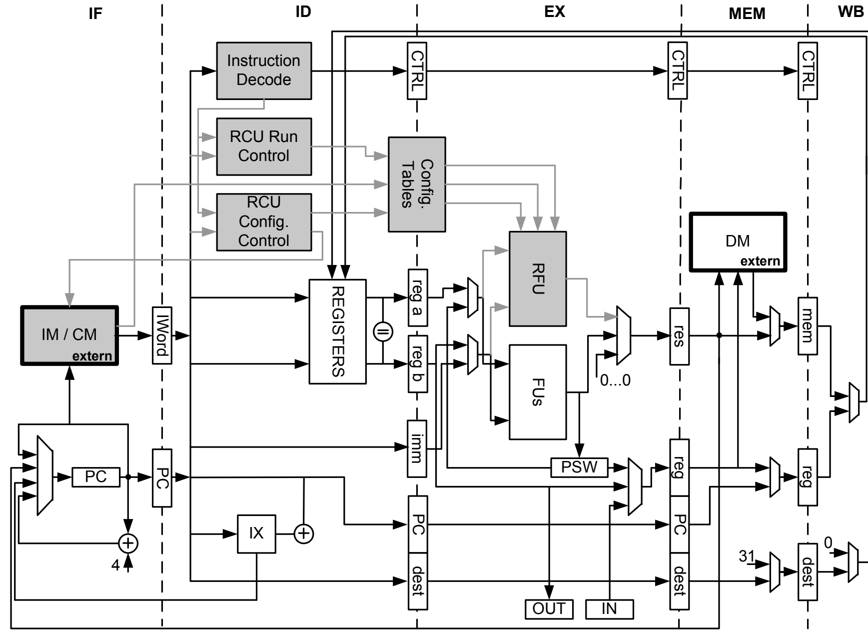
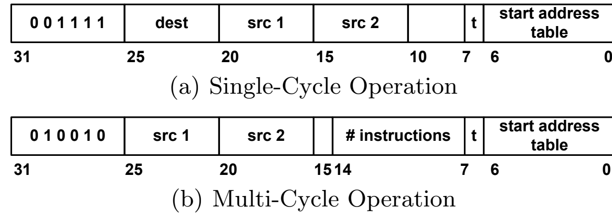


Fig. 12. RISC processor with integrated RFU

In order to configure and program the RFU, three new instructions, one for configuration and two for execution, are added to the instruction set. The new instructions are passed from the instruction decode unit directly to the configuration control unit, which either performs the loading of the configuration memory or passes the instructions to the run control logic. By specifying the start address of the configuration data, block size, configuration table and table entry, the configuration data is downloaded by the configuration control unit autonomously to the dedicated configuration table. In the meantime the processor can continue to execute its program.

For the operation of the RFU, two instructions are available. The first operation performs a single-cycle operation while the second operation can be used for specifying a multi-cycle operation. Figure 13 shows the format of the two instructions. The multi-cycle instruction can be used to initiate an autonomous execution of the RFU over several clock cycles. In order to avoid pipeline hazards, the multi-cycle instruction does not contain a destination address for the result of the operation. Therefore a singlecycle instruction has to be used for writing the result to the register file. A more detailed description of the reconfiguration and system integration can be found in [13].



(a) Single-Cycle Operation

(b) Multi-Cycle Operation

Fig. 13. Operation Instructions

6 Results

A test system was synthesized using Synopsys' Design Analyzer with a $0.25\mu m$ 1P5M CMOS standard cell technology. The test system consists of the RFU and the modified DLX processor as presented in the last section. For larger memories like the *LUT Module* inside the *ECM* block and the configuration tables of the RFU control logic, RAM macro cells have been employed. All area values are normalized to the area of an eight bit multiplier without pipelining.

Table 1. Reed-Solomon Decoder Structure

Module	**Area** (normalized)	**Freq.** $[MHz]$
CRC En-/Decoder	9.9	205
RS Encoder	11.4	209
RS Decoder	95.0	59
Total Area	116.3	-

Table 2. Synthesis Results of the *ECM* Block

Module	**Area** (normalized)	**Freq.** $[MHz]$
Block A	9.3	140
Block B	10.0	172
Other Blocks	3.0	694
ECM	22.3	137

Synthesis and performance results of the reference designs are presented in table 1. For CRC computation a hardware architecture based on parallel CRC-8 encoding/decoding using Galois Field arithmetic was used. Values for Reed-Solomon encoder and decoder base on a RS(255,239) code using the architecture presented in section 3. The synthesis results for the *ECM* block can be found in table 2. The *ECM* block requires only about 20% of the area of the reference

design. Even if the *Common Resource* block and the *LUT Module* of the RFU are counted to the area of the reconfigurable design, hardware savings of up to 28% in comparison with the standard implementation presented in section 3 can be achieved. These hardware savings ease the design of area efficient ASIC solutions for mobile terminals when using the RFU for error detection and correction instead of standard implementations.

Table 3. Throughput of the Reference Design and the Reconfigurable Architecture

Application	Recon. Arch.	Ref. Design
RS(255,239) enc.	735.8 Mbps	1568.5 Mbps
RS(255,239) dec.	372.8 Mbps	220.7 Mbps
CRC8 enc./dec.	2512.3 Mbps	13113.7 Mbps

Throughput values for both architectures are given in table 3. For encoding RS(255,239) codes and CRC8 encoding/decoding the reference architecture is faster than the reconfigurable design, but for RS(255,239) decoding a speed-up of 1,68 could be achieved. All throughput rates are more than sufficient in relation to the data rates required for mobile terminals. Taking the actual WLAN standard IEEE 802.11a as a reference, data rates of only 42 Mbps are required at the MAC-layer [6].

Table 4. Synthesis Results for the RFU

Module	Area (normalized)	Freq.$_{[MHz]}$
Basic CPU	44.3	199
ECM Block	22.3	137
AES Block	19.1	138
LUT Module	29.0	588
Common Res.	20.4	201
RFU Control	37.3	244
Total Design	170.0	98

Synthesis results for the DLX processor with the RFU and all components of the RFU can be found in table 4. The chip area of the complete design has a normalized value of about 170. Only 13% of the total chip area are required by the *ECM* block. The area fraction of the RFU is about 74.1% of the overall area. A huge part of the RFU is constituted by the configuration tables (648 bytes) and the look-up tables in the *LUT module* (1024 bytes). The memory blocks add up to a normalized area of about 66 which is about 39% of the overall area. To maximize the utilization of these memory blocks, the input and output ports of the look-up tables are directly accessible and thus can be used as additional memory for the processor.

7 Conclusion

In this paper a function-specific dynamically reconfigurable architecture for error detection and error correction has been presented. The architecture offers two levels of (re)configuration; on the one hand it can be configured to perform several algorithms (e.g. CRC, Reed-Solomon Codes with variable code parameters), on the other hand it reuses hardware components by means of dynamic reconfiguration. Synthesis and performance results have proved that the architecture offers an attractive alternative to standard implementations, in particular as its hardware resources can be utilized by the processor for additional tasks.

Acknowledgement

The work was supported by the German Research Foundation (DFG - Deutsche Forschungsgemeinschaft) within the special research program *Reconfigurable Computer Systems* under GL 155/25.

References

1. Advanced Encryption Standard (AES), November 2001. Federal Information Processing Standards Publication 197.
2. Hyunman Chang and Myung H. Sunwoo. Design of an Area Efficient Reed-Solomon Decoder ASIC Chip. *IEEE Workshop on Signal Processing Systems*, pages 578–585, October 1999.
3. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.
4. Yuh-Tsuen Horng and Shyue-Win Wei. Fast Inverters and Dividers for Finite Field $GF(2^m)$. *IEEE Asia-Pacific Conference on Circuits and Systems*, pages 206–211, December 1994.
5. Huai-Yi Hsu and An-Yeu Wu. VLSI Design of a Reconfigurable Multi-mode Reed-Solomon Codec for High-Speed Communication Systems. *Proceedings of the IEEE Asia-Pacific Conference on ASIC*, pages 359–362, August 2002.
6. Jangeun Jun, Pushkin Peddabachagari, and Mihail Sichitiu. Theoretical Maximum Throughput of IEEE 802.11 and its Applications. In *NCA '03: Proceedings of the Second IEEE International Symposium on Network Computing and Applications*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.
7. Dong-Sun Kim, Jong-Chan Choi, and Duck-Ji Chung. Implementation of High-Speed Reed-Solomon Decoder. *42nd Midwest Symposium on Circuits and Systems*, 2:808–812, August 1999.
8. P. Kitos, G. Theodoridis, and O. Koufopavlou. An efficient reconfigurable multiplier architecture for Galois field $GF(2^m)$. *Microelectronics Journal*, 34(11), November 2003. Elsevier.
9. Hanho Lee, Meng-Lin Yu, and Leilei Song. VLSI Design of Reed-Solomon Decoder Architectures. *Proceedings of the IEEE International Symposium on Circuits and Systems*, 5:705–708, May 2000.

10. Edoardo D. Mastrovito. VLSI Designs for Multiplication over Finite Fields $GF(2^m)$. In *AAECC-6: Proceedings of the 6th International Conference, on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 297–309, London, UK, 1989. Springer-Verlag.
11. H. Michael Ji and Earl Killian. Fast Parallel CRC Algorithm and Implementation on a Configurable Processor. *IEEE International Conference on Communications*, 3:1813–1817, April 2002.
12. Christof Paar and Martin Rosner. Comparison of Arithmetic Architectures for Reed-Solomon Decoders in Reconfigurable Hardware. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 219–225, Los Alamitos, CA, April. IEEE Computer Society Press.
13. Thilo Pionteck, Thorsten Staake, Thomas Stiefmeier, Lukusa D. Kabulepa, and Manfred Glesner. Design of a Reconfigurable AES Encryption/Decryption Engine for Mobile Terminals. *Proceedings of the 2004 IEEE International Symposium on Circuits and Systems*, 2:545–548, May 2004.
14. Tenkasi V. Ramabadran and Sunil S. Gaitonde. A Tutorial on CRC Computations. *IEEE Micro*, 8(4):62–75, July 1988.
15. Sourav Roy, Wolfgang Wilhelm Martin Bücker, and B.S. Panwar. Reconfigurable Hardware Accelerator for a Universal Reed Solomon Codec. *Proceedings of 1st IEEE International Conference on Circuit and Systems for Communication*, pages 158–161, June 2002.