Analysis of Reed Solomon error correcting codes on reconfigurable hardware

Master's Thesis

by

Anne Franssens

Committee:
Prof. dr. ir. G.J.M. Smit
Dr. ir. G.K. Rauwerda (Recore Systems)
ir. P.T. Wolkotte

University of Twente, Enschede, The Netherlands
September 9, 2008

Abstract

This thesis describes the advantages and disadvantages of using architectures consisting of multiple different (types of) processors. Such architectures are better known as heterogeneous architectures. The research is focussed on determining processing power and energy efficiency, based upon the mapping of a Reed Solomon (RS) error correcting decoder on a reconfigurable architecture. Parts of the RS decoder that are not implemented, will be executed on a General Purpose Processor which can communicate with the reconfigurable hardware, which is better known as a heterogeneous architecture. The GPP used, is an ARM, and the reconfigurable architecture used is a Montium. The RS algorithms are implemented to derive conclusions as to which architecture is favorable for implementation of the different RS sub-blocks, and how processing power and energy efficiency relate to other architectures.

The research focusses on RS applied for the Digital Media Broadcasting (DMB) standard. This standard prescribes several parameters of the RS coding, such as packet-size(n), and the amount of parity symbols per packet (k). To facilitate in the reception and processing of DMB, the algorithms are tailored to fit its requirements.

To be able to derive conclusions, two sub-blocks of RS are implemented on the Montium. These are the syndrome calculator and the error locator. The syndrome locator has to be executed for each incoming RS data-packet, and determines if the packet is error-free or not. If it is not, the other RS blocks are executed which determine the errors and their locations. The error locator is the part which calculates the locations of the errors. Together, the syndrome calculator and the error locator use almost 90% of the average calculation time on an ARM, which makes them the most computationally intensive parts. To get acquainted with the hardware and familiar with its tools, also an RS encoder has been designed. The implementation and mapping of these three algorithms is explained in this thesis.

Conclusions are based on a comparison of the Montium to other architectures. These are the ARM (GPP), the Intel® Pentium® M (GPP) and a Xilinx IP Core. The Montium performs well on the implemented decoder blocks, even though its clock frequency is relatively low. It is capable of performing the calculations faster than the ARM and the Pentium, and is much more energy efficient. The Xilinx core is much faster than the Montium, but comparison on energy performance is difficult. When flexibility on reconfiguration level is an issue, the Montium is the best choice.

Contents

List of Figures

List of Tables

List of Source Codes

Acknowledgements

This thesis concludes my Master-study Computer Science, track Embedded Systems, at the University of Twente. I wrote this thesis at the Computer Architecture Design & Test for Embedded Systems (CADTES) chair, which was later changed to Computer Architecture for Embedded Systems (CAES).

I want to thank Gerard Smit for offering this interesting assignment, and his influence and patience concerning my work. I also want to thank the other members of the committee, ir. Pascal Wolkotte and dr.ir. Gerard Rauwerda for their continuous effort, guidance and reviews which made this thesis possible.

I would also like to thank Prof.dr.ir. Thijs Krol for his private lectures on the basics of Reed Solomon, and ir. Robert de Groote for his lectures on Galois theory. Furthermore I would like to thank Kenneth Rovers for giving me an introduction to Reed Solomon.

Gerard Smit has given me the opportunity to really feel like a part of CADTES/CAES, for which I am very grateful. I have enjoyed the Chameleon meetings, the coffee breaks, the strolls and the dinners, and have always felt welcome. I would also like to thank the staff and the Ph.D. students for the things I have learned from them and the conversations and fun we have had.

Many thanks also to my friends, other colleagues and everyone who supported me during my past years at the University of Twente. Last but not least I wish to thank my parents. They have always supported me in any way possible, and still do. I would like to thank them for their patience and support, and I am proud to present my thesis in front of them.

Anne Franssens
Enschede, 2008

1 Finite Fields

The mathematical basis of Reed Solomon is based on the theory of finite fields. Finite fields are based on a prime number p and consist of p elements. As their existence and importance have been discovered by Éveriste Galois [10], they are denoted $GF(p^m), m \in \mathbb{Z}^+$. Perhaps the most characterizing feature of Galois Fields is that they are ring-structured. They are finite sets, closed under addition and multiplication; the results of these arithmetic operations always result in values that are members of the field.

Galois Fields consist of a field of p^m elements. For the construction of Reed-Solomon codes, symbols from the extension field $GF(2^m)$ are used.

The extension field consists of the values 0 and 1, and additional values that are powers of a symbol $\alpha^i \in p^m$. This results in an infinite set F. To create a finite subset a closing condition is needed which is defined by the irreducible polynomial $\alpha^{(2^m-1)} + 1 = 0$. The set F' is thereby reduced to:

$$F' = \{0, \alpha^0, \alpha^1, ..., \alpha^{2^m-2}\} \tag{1.1}$$

The actual field consists of elements that are a power of a primitive polynomial. Such a polynomial is obtained by selecting an irreducible polynomial from the set of $GF(p^m)$. This set contains every polynomial of degree m and arithmetic operations are done modulo p. Then consider a root α of this irreducible polynomial. The powers of α, ranging from α to $\alpha^{p^{m-1}}$, are the possible polynomials and thus the elements of the field. All arithmetic operations are performed under the modulo of the generator polynomial ($G(X)$). If a result is out of the field, taking the modulo maps it back to the original field.

$$-\alpha^y \equiv \alpha^y \tag{1.2}$$

$$b\alpha^y \pmod{b} \equiv 0 \tag{1.3}$$

This ensures that the result is always a member of the field. Primitive polynomials are usually depicted as a single integer number. For example, 37, which in binary is 100101, represents the polynomial $f(\alpha)$:

$$\alpha^5 + \alpha^2 + 1 \tag{1.4}$$

When converting to and from the field, the same primitive polynomial must be used.

1.1 Addition and subtraction

Addition and subtraction in a Galois Field is done modulo p. For polynomials this means that they can be added and subtracted, but the coefficients of the resulting polynomial must be taken modulo p. This also implies that there is no need for a carry-flag or bit. Because of the modulo, coefficients are limited to a number of p. As messages in hardware are composed of bits, p is 2 by standard. The polynomials then are no more than strings of bits. Adding or subtracting their strings whilst taking each resulting bit modulo p is a simple but effective XOR operation.

Example:

$101 \rightarrow \alpha^2 + ... + 1$
$011 \rightarrow ... + \alpha^1 + 1$

$$\begin{array}{l} \alpha^2 + 1 \\ \underline{\alpha^1 + 1 \quad +} \\ \alpha^2 + \alpha^1 + 2 \end{array} \qquad \begin{array}{l} \alpha^2 + 1 \\ \underline{\alpha^1 + 1 \quad -} \\ \alpha^2 + \alpha^1 \end{array}$$

$$\alpha^2 + \alpha^1 + 2 \pmod 2 \equiv \alpha^2 + \alpha^1$$

$$\alpha^2 + \alpha^1 \longrightarrow 110 = 101 \; XOR \; 011$$

1.2 Multiplication

Multiplication in Galois Fields is more difficult than addition or subtraction. It is defined as multiplying the polynomials. In order to let the result remain in the field, a modulo with the generator polynomial is then performed. The following formula specifies the multiplication of $f(\alpha)$ and $g(\alpha)$ of degrees n and m, and coefficients $f_i, g_j \in \{0, 1\}$:

$$f(\alpha) \cdot g(\alpha) = \left(\left(\sum_{i=0}^{n} f_i \left(\sum_{j=0}^{m} g_j \alpha^{i+j} \right) \right) \mod p \right) \mod G(\alpha) \tag{1.5}$$

Example:

Let $f(\alpha) = \sum f_i \alpha^i$ *of degree* $n = 5$ *and* $g(\alpha) = \sum g_i \alpha^i$ *of degree* $m = 4$, *and let* $f(\alpha) = 58$, $g(\alpha) = 29$ *and the generator* $G(\alpha) = 285$, *so*

$$f(\alpha) = \alpha^5 + \alpha^4 + \alpha^3 + \alpha$$
$$g(\alpha) = \alpha^4 + \alpha^3 + \alpha^2 + 1$$
$$G(\alpha) = \alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + 1$$

$f(\alpha) \cdot g(\alpha)$
$= (\alpha^5 + \alpha^4 + \alpha^3 + \alpha) \cdot (\alpha^4 + \alpha^3 + \alpha^2 + 1)$
$= \alpha^9 + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 + \alpha^7 + \alpha^6 + \alpha^5 + \alpha^3 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^1$
$= \alpha^9 + 2\alpha^8 + 3\alpha^7 + 2\alpha^6 + 3\alpha^5 + 2\alpha^4 + 2\alpha^3 + \alpha^1 \pmod 2$
$= \alpha^9 + \alpha^7 + \alpha^5 + \alpha^1$

The result of the multiplication modulo $G(\alpha)$:
$= \alpha^9 + \alpha^7 + \alpha^5 + \alpha^1 \pmod{G(\alpha)}$

Calculating the modulo by repeated longdivision with $G(\alpha) \cdot \alpha^y$ results in:

$$\begin{array}{l} \alpha^9 + \alpha^7 + \alpha^5 + \qquad\qquad \alpha^1 \\ \underline{\alpha^9 \qquad\quad + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^1 \quad -} \\ \qquad\quad \alpha^7 \qquad\quad + \alpha^4 + \alpha^3 \end{array}$$

$\alpha^7 + \alpha^4 + \alpha^3 = 10011000 = 152$
So $58 \cdot 29 = 152$

As a real Galois modulo operator is difficult to implement on hardware, and polynomial multiplications are computationally expensive, other algorithms have been discussed in literature that use different approaches.

Direct implementation

The direct implementation is the most standard method for implementing a multiplication. It uses the basic multiplication technique of equation 1.5. It recognizes that the modulo can be substituted for a maximum of one division. The remainder is the result of the multiplication. Note this only holds if the generator polynomial is larger than p^m, which is always the case [15].

If the degree of the p^m polynomial is m, the polynomials in the field also have a maximum degree of m. Therefore, it can be easily concluded that multiplying two polynomials of the field and dividing that result by the generator polynomial (with degree $\geq m$), results in a polynomial with a maximum degree of m. This is why after the multiplication, the modulo operation consists in fact of at most one divisions. Taking the remainder of this division is the final result of the modulo operation.

This method has the advantage that the number of steps that need to be taken to acquire the result is always known. The drawback is that multiplying and dividing two polynomials is computationally expensive.

Multiplication table

An obvious but less elegant solution is the use of a multiplication table. This is a table in which all possible multiplications and its outcomes are entries. In the case of Reed Solomon, the parameters are bytes, thus having $x = 256$ possible values. A multiplication table would then consist of $\frac{1}{2}x^2 = 16384$ entries, or 16kB.

	α^0	α^1	α^2	α^3
α^0	α^0	α^1	α^2	α^3
α^1	α^1	α^2	α^3	α^0
α^2	α^2	α^3	α^0	α^1
α^3	α^3	α^0	α^1	α^2

Table 1.2: Multiplication LUT with $p = 2, m = 4$

Logarithm table

The method with the logarithmic lookup table (LUT) requires less memory. The theory is based upon the mathematical expression: $\log(a \cdot b) = \log(a) + \log(b)$.

Every element of the field is expressible as a power of the primitive polynomial α (section 1).

$$(\alpha^i) \cdot (\alpha^j) = \alpha^{i+j \mod p^m - 1} \tag{1.6}$$

Using this formula, multiplications can be calculated by knowing the exponent i and j of the values that need to be multiplied. Since $\alpha^i = x$, i can be referred to as the "discrete logarithm" of x. By constructing the table for these logarithmic values, the multiplication can be reduced to a repeated look-up operation in this table.

There needs to be at least a table containing the log of every element of the set F'. In that table the log of a and b can be easily selected and added. The result can be easily read from a second table containing the exponents. Another option would be to reversely look it up in the log table, by finding the index of the value that equals the result. Because the tables can be determined based upon $G(\alpha)$ they can be loaded into memory and do not have to be calculated. Therefore, the approach using two tables uses more memory, but discards the need for a cycle-consuming reverse lookup. A simple index-based lookup would then suffice.

Generating the LUT

Generating the lookup table is a procedure that only has to be done once for a specific field. The lookup table has to define the discrete logarithm of all of the values $[0, 1, ..., p^m - 1]$. More formally: take a primitive element α from a finite field F. Define β of F as its discrete logarithm. This leads to the following simple statement: find the smallest $m \in F$ for which $\alpha^m = \beta$. A practical approach to generate the table is to add the coefficients of the α values, $\mod p^m - 1$. Table 1.2 contains an example multiplication table. The inverse table (exponent table) is simply an index-value switched inverse of the logarithm table.

2

Introduction to Reed Solomon

Reed-Solomon (RS) codes were firstly described in 1960 by Irving S Reed and Gustave Solomon [15]. The theory is based upon the theory of Bose-Chaudhuri-Hocquenghem (BCH) codes. RS codes are members of the Forward Error Correcting codes (FEC). In principle they add redundant parity symbols to the end of a message. When the Signal to Noise Ratio (SNR) is low, bits could get flipped during transmission. By using the redundant parity symbols, the original message can, up to a certain point, be recovered (Figure 2.1). This thesis will be based on the RS specification of the Digital Media Broadcasting (DMB) standard. For mobile communication, DMB channels are used that support up to a maximum of 1.152Mbps [18]. The high data-rate DVB-H specification supports up to a maximum of 15Mbps. The properties of the used RS code is defined by the used standard, and is discussed in section 2.1.

2.1 Applicational and mathematical properties

RS codes are non-binary cyclic codes [15]. This means that $p \geq 2$. When applied on digital systems p is chosen to be 2, so its values can be expressed in a binary representation. The symbol size is m bits where m is an integer and at least 2. RS codes are denoted as RS(n, k) where k is the number of symbols that is encoded and n is the total number of symbols in the encoded block. Figure 2.2 illustrates the structure of an RS codeword.

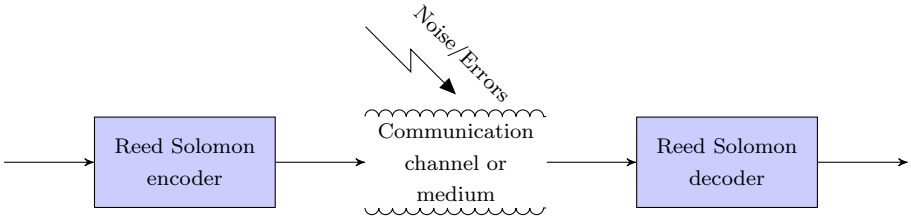


Figure 2.1: Typical Reed Solomon setup

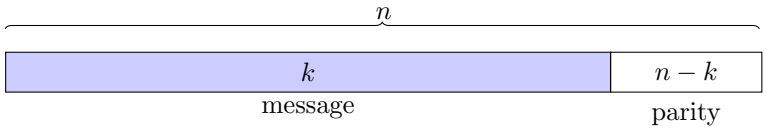


Figure 2.2: Reed Solomon codeword

SYMBOL1	SYMBOL2	SYMBOL3	SYMBOL4	SYMBOL5	SYMBOL6	SYMBOL7	SYMBOL8	SYMBOL9	SYMBOL10
OK	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	OK

Figure 2.3: Datablock containing an error of 8 up to 64 bits (m=3)

The n and k must satisfy the following equation:

$$0 < k < n < p^m - 1 \tag{2.1}$$

The number of parity symbols that are added can be obtained by $n - k = 2t$, where t denotes the number of symbol errors this specific code can correct. t can then be expressed as

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor \tag{2.2}$$

When evaluating codes, the minimum distance between codewords should be as large as possible. For non-binary codes this is defined by the Hamming-distance d; the number of symbols in which two codewords differ. For RS codes this minimum distance is

$$d_{min} = n - k + 1 \tag{2.3}$$

Because of equation 2.2 this means that

$$d > 2t \tag{2.4}$$

Burst errors

RS codes are designed for correcting errors on noisy channels. This is partially due to the fact that noisy channels (e.g. low SNR) create so called "burst errors". A burst error is a sequence of continuous erroneous bits that possibly affect multiple symbols. The DMB standard uses an RS(204, 188) code, where each symbol is made up of $m = 8$ bits (1 byte). Equation 2.2 states that this code can correct 8 erroneous symbols in a block of 204. If a burst error occurs of for example 57 bits, it would corrupt at least $\lceil \frac{57}{m} \rceil = 8$ symbols. The power of non-binary codes lies therefore in the fact that if a symbol contains an error it might as well be completely erroneous. As noisy channels usually create burst errors, a large number of bit-errors only affect a few symbols. Figure 2.3 shows how a continuous 64 bits error can corrupt 8 symbols, or how 8 non-continuous bit errors can cause the same 8 symbols to be corrupted.

Interleaving can also be used to further reduce error-rates. This is particulary useful when burst-erroneous behaviour is expected. After RS encoding the data, the resulting symbols are interleaved. If a burst error occurs, several

small parts of a large number of blocks are malformed, thus increasing the chance of every packet being decodable. An example, applied in the Compact Disc standard, is given in [12].

Galois Fields and RS codes

Reed solomon codes are codes that "exist" in a certain Galois Field. To encode and decode messages, $G(X)$ is needed. The generator polynomial is of the following form:

$$G(X) = g_0 + g_1X + g_2X^2 + ... + g_{2t-1}X^{2t-1} + X^{2t} \tag{2.5}$$

The degree of G equals the number of parity symbols. Because G is of degree $2t$, there are exactly $2t$ successive powers of α that are roots.

$$G(X) = (X - \alpha^{m_0})(X - \alpha^{m_0+2})...(X - \alpha^{m_0+2t}) \tag{2.6}$$

m_0 is usually chosen to be zero or one. Note that m_0 is to indicate that any $2t$ successive powers of α can be used. In this thesis, m_0 is chosen to be 1.

The more formally written equation of $G(X)$:

$$G(X) = \prod_{i=0}^{2t-1} (X - \alpha^{m_0+i}) \tag{2.7}$$

2.2 Encoder structure

Encoding cyclic RS codes is done in the same way as encoding binary codes [15]. To be able to perform mathematical operations on codewords and code blocks, they are represented in a polynomial form. This happens in the same way as with the generator polynomial. A message (data) of for example α^1, α^6, α^2 is represented by a polynomial

$$D(X) = \alpha^1 + \alpha^6X + \alpha^2X^2 \tag{2.8}$$

As RS codes are systematic codes, the code blocks consist of a message D and parity P, that together form the codeword C. The message part of the codeword is the actual message D, and therefore remains untouched. The parity has to be calculated.

More formally, $C(X)$ consists of $P(X)$ and $D(X)$. To create C, D has to be placed in the rightmost part of C, or, multiplied by X^{n-k}. This means right-shifting D by $n - k$ positions.

The P can be calculated by dividing the right-shifted D by the generator polynomial G and taking its remainder.

$$P(X) = X^{n-k}D(X) \bmod G(X) \tag{2.9}$$

Combining the expressions for $P(X)$ and $D(X)$ yield the expression for the codeword

$$C(X) = P(X) + X^{n-k}D(X) \tag{2.10}$$

Formal encoder structure

This section will describe the encoder structure in a more formal mathematical way.

Define a message (data) polynomial, and its encoded codeword polynomial as

$$\begin{aligned} D(X) &= d_{k-1}X^{k-1} + d_{k-2}X^{k-2} + ... + d_1X + d_0 \\ C(X) &= c_{n-1}X^{n-1} + c_{n-2}X^{n-2} + ... + c_1X + c_0 \end{aligned} \tag{2.11}$$

Each codeword is a multiple of the generator polynomial $G(X)$. The 2^t consecutive powers of $\alpha^{m_0}, \alpha^{m_0+1}, .., \alpha^{m_0+2t-1}$ are roots of $G(X)$, and because C(X) is a multiple of G(X), they are roots of C(X) as well.

Encoding $D(X)$ into a $C(X)$ polynomial requires the calculation of a parity polynomial $P(X)$. If $Q(X)$ and $P(X)$ are seen as a quotient and remainder, the equation for $C(X)$ can be written as

$$X^{n-k}D(X) = Q(X)G(X) - P(X) \tag{2.12}$$

Referring to the previous section of this chapter; the right-shifter polynomial $D(X)$ is divided by $G(X)$, with $P(X)$ as a remainder. $C(X)$ can then be derived as follows

$$Q(X)G(X) = X^{n-k}D(X) + P(X) = C(X) \tag{2.13}$$

This also shows that $C(X)$ is a multiple of $G(X)$ which is a property that simplifies the decoding process later on. The codeword $C(X)$ is then of the form

$$(c_{n-2}, c_{n-1}, ..., c_1, c_0) = (d_{k-1}, d_{k-2}, ..., d_1, d_0, p_{n-k-1}, p_{n-k-2}, ..., p_1, p_0) \tag{2.14}$$

where the negative values of p can be taken absolute because of the the property given in equation 1.2.

2.3 Decoder structure

Decoding RS codes is far more complicating than encoding them. When a transmission channel is noisy, it leads to a compromised message which has been affected by a certain error $E(X)$. As flipping bits is essentially the same as adding them (without carry), it can be concluded that the received word is actually the codeword with an added error (Figure 2.4).

$$R(X) = C(X) + E(X) \tag{2.15}$$

in $R(X)$ there are $2t$ unknowns. In the more easy binary versions of BCH codes, the only required information is the location at which the error has occurred. By knowing the error location and the current value of the bit, it is simply a matter of flipping that bit. In RS codes not only the location has to be known, but also the error value. Since there are $2t$ unknowns, $2t$ equations are needed for the solution.

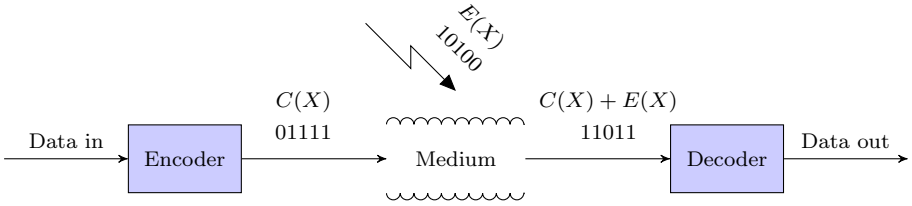


Figure 2.4: Example of introduced error vector

$E(X)$ is defined as the error polynomial:

$$E(X) = R(X) - C(X) = e_0 + e_1X + e_2X^2 + ... + e_{n-1}X^{N-1} \tag{2.16}$$

where $e_i = r_i - c_i$ is a symbol of the Galois field.

Assume that $R(X)$ has v errors in positions $i_1, i_2, .., i_v$. $E(X)$ can then be written as

$$E(X) = e_{i_1}X^{i_1} + e_{i_2}X^{i_2} + ... + e_{i_v}X^{i_v} \tag{2.17}$$

Using the $E(X)$, the error locations and the error values can be calculated. Define $\beta_k = \alpha^{i_k}, k \in \{1..v\}$ to be the error location numbers. The actual error values are

$$e_{i_1}, e_{i_2}, ..., e_{i_v} \tag{2.18}$$

The next sections will explain how these values can be calculated.

Syndromes

To verify whether a received word $R(X)$ is a valid codeword, the syndrome of $R(X)$, referred to as S, needs to be calculated. If $R(X)$ is indeed a valid codeword, the syndromes will be zero. An RS code has $2t$ syndromes, $s_j, j \in (1, ..., 2t)$.

As the structure of the code is based upon (see equation 2.2):

$$C(X) = D(X)G(X) \tag{2.19}$$

It can be seen that the roots of $G(X)$ are also roots of $C(X)$. As $E(X)$ is supposed to be zero, the condition $C(X) = R(X)$ should hold, meaning that the roots of $G(X)$ are also roots of $R(X)$. Therefore, syndromes s_i are the outcomes of $s_i = R(roots_{G(X)})$. If they are not zero, the $R(X)$ is not a valid codeword. More formally, the syndrome values can be expressed as

$$s_j = R(\alpha^j), j \in (1, ..., 2t) \tag{2.20}$$

Since equation 2.19 holds, we can derive these relations:

$$R(\alpha^j) = C(\alpha^j) + E(\alpha^j) = E(\alpha^j) \tag{2.21}$$

$$s_j = R(\alpha^j) = E(\alpha^j) \tag{2.22}$$

$$s_j = e_{i_1}\beta_1^j + e_{i_2}\beta_2^j + ... + e_{i_v}\beta_v^j \tag{2.23}$$

When the sequence of the 2t syndrome equations is expanded, we obtain

$$\begin{aligned} s_1 &= e_{i_1}\beta_1 + e_{i_2}\beta_2 + ... + e_{i_v}\beta_v && (2.24)\\ s_2 &= e_{i_1}\beta_1^2 + e_{i_2}\beta_2^2 + ... + e_{i_v}\beta_v^2 && (2.25)\\ \vdots &= \vdots \\ s_{2t} &= e_{i_1}\beta_{i_1}^{2t} + e_{i_2}\beta_{i_2}^{2t} + ... + e_{i_v}\beta_{i_v}^{2t} \end{aligned}$$

The syndromes can be expressed as a syndrome polynomial, where the s_j values are the coefficients:

$$S(X) = s_1 + s_2X + ... + s_{2t}X^{2t-1} \tag{2.26}$$

Error locations

To find the location of the errors the error locator polynomial $\Lambda(X)$ is derived from the syndrome equations. The roots of $\Lambda(X)$ are the inverses of the error locators β_i.

$$\Lambda(X) = \prod_{j=1}^{2t}(1 - \beta_j X) = 1 + \lambda_1 X + \lambda_2 X^2 + ... + \lambda_{2t}X^{2t} \tag{2.27}$$

If there are v distinct non-zero λ coefficients where $v \leq t$, the codeword is correctable. The number of errors v in the codeword is also known at this point.

An efficient algorithm for calculating $\Lambda(X)$ is described by Berlekamp-Massey [15]. When $\Lambda(X)$ is calculated, the error locations are the roots of Λ. This means the α values for which $\Lambda(\alpha^i) = 0, i \in \{1..2^m\}$.

Error values

The error values e_i can be found by Forney's algorithm. Forney's algorithm requires the calculation of $\Lambda(X)$ a priori. The error magnitude polynomial is defined as follows:

$$\Omega(X) = (1 + S(X))\Lambda(X) \tag{2.28}$$

Forney's algorithm defines the error values as:

$$e_{i_l} = -\frac{X_l\Omega(X_l^{-1})}{\Lambda'(X_l^{-1})} \tag{2.29}$$

where $\Lambda'(\gamma)$ is the derivative of $\Lambda(\gamma)$.

Once the error locations and error values are known, the error value can be added to $R(X)$ at the given location to correct the received word. Forney's algorithm is not used in this research. For further documentation we refer to [15].

3

Reconfigurable heterogeneous architectures

The demand on today's embedded architectures are high. It is impossible to achieve the highest level of performance, energy-efficiency, real-time properties, reliability and flexibility at the same time. Therefore, different structured architectures have been developed, each with its own unique trade-off of properties.

Because all architectures' trade-offs are different, their ability to be applied in different types of systems is limited.

Target devices, such as Mobile devices; mobile phones, PDAs and multimedia players, have their own set of requirements. They try to achieve a maximum of performance but are restricted by energy-efficiency requirements which keep the battery from draining within the hour. Computational intensive image-render farms, on the other hand, are not bound by energy usage and therefore use architectures that maximize performance.

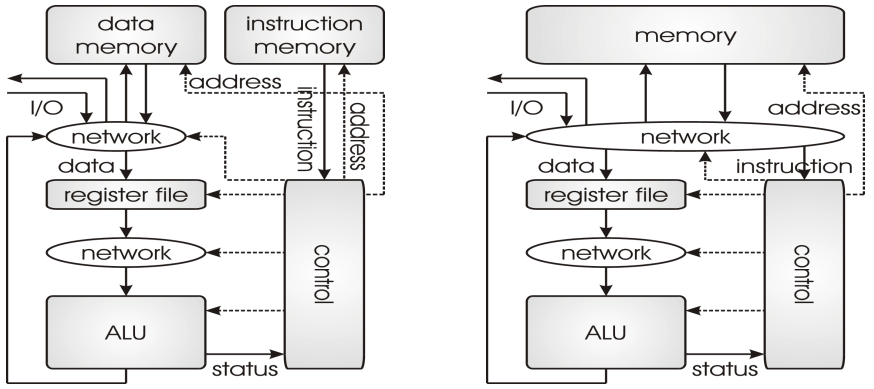
The algorithm domain also limits the designer´s architectural choice. Some algorithms are computational intensive, memory intensive or control intensive. To achieve maximum performance while staying in bound of the target-device requirements, choices have to be made. Mapping computationally intensive algorithms on an architecture which design favored memory intensive algorithms, could lead to poor performance.

This chapter discusses both the different types of algorithm and the different architectures available.

3.1 Common architectures

There are several architectures available with specific properties. The four most common types are:

- general-purpose processors (GPPs)
- digital signal processors (DSPs)
- Field programmable Gate Arrays (FPGAs)
- application specific integrated circuits (ASICs).



(a) Harvard architecture

(b) Von Neumann architecture

Figure 3.1: Harvard and Von Neumann architectures [11]

General-Purpose Processor

The GPP is best known for its application in Personal Computers (PCs), although it is also widely used in embedded systems. The instruction set of the GPP is very basic, but allows for any algorithm to be implemented. The downside is that programming complex algorithms on instruction-level (assembly) is hard. Due to the wide extent in which the GPP is used, many high-level compilers and languages are available for the GPP. This makes programming for the GPP a breeze compared to programming devices in assembly language. Disadvantages of the GPP are its relatively poor performance and its low energy-efficiency due to its optimization for flexibility.

GPPs were designed using only one memory component. This memory contained both instructions and data. In this architecture, programs could span the entire memory, thus without further restrictions. The drawback of this architecture (Von Neumann architecture [20], Figure 3.1b) is that both data and instructions reside in the same, usually single channeled, memory. Performing operations on data requires at least two cycles because they can not be loaded at the same time. Furthermore, ALU speeds increased much faster than memory speeds therefore resulting in blocking reads from memory. This bottleneck is referred to as the *Von Neumann bottleneck*. A possible solution to (part) of this problem is splitting the memory into an instruction- and a data part (Harvard Architecture, figure 3.1a). Another method for increasing the fetching of data or instruction from memory is using caches. General improvements to the GPP are the use of pipelining, multiple data memories (Dual Harvard architecture) and extending its functionality by adding function units (e.g. Floating Point unit).

Digital Signal Processor

DSPs are optimized for digital signal processing. A DSP often assists a host GPP by acting as a real-time stream processor. The instruction set of a DSP is usually larger than the instruction set of a GPP, because it is extended with

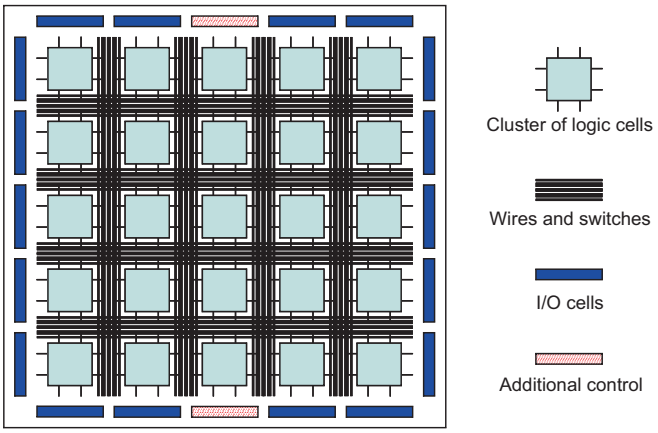


Figure 3.2: Structure of an FPGA [11]

additional specialized DSP instructions. Very Long Instruction Word DSPs are available, packing multiple regular instructions into a very long instruction word. Energy efficiency and performance of the DSP are higher than the GPP. Reasons for this, include: more combinatorial hardware instead of control hardware, and no traditional time-sharing.

Field Programmable Gate Array

The FPGA (Figure 3.2) is an integrated circuit which consists of a two-dimensional array of logic-cells. Both the logic-cells and the interconnecting switches and wires can be programmed. Complex algorithms can be executed by combining switches, paths and logic-cells to form complex structures (e.g. adders and shifters). The outside of the logic-cell matrix consists of I/O cells and additional control, to make communication to the outside world possible.

The programmability of an FPGA is fairly hard compared to a GPP. Because of the structure of the FPGA, programs do not only have to be compiled. It is also required that a synthesis step is made which maps the program onto the hardware. This is in contrast with the GPP, which hardware does not need to be adapted to the program. Programs can be written in a Hardware Description Language (HDL) such as VHDL[3] and Verilog [2]. After programming, the synthesization tool creates a mapping onto the specific target FPGA. This takes a considerable amount of time, due to the optimizations the tool has to exploit for the FPGA. Due to the fact that every switch, path and logic-cell-structure has to be bit-wise configured, the amount of configuration data is very large (order of Mb). Configuring an FPGA is therefore relatively slow, and runtime reconfiguration takes a considerable amount of time. Modern FPGAs often have the ability to be partially reconfigured, which is an advantage if only a small part of the algorithm changes.

Where the GPP needs consecutive cycles to perform repetitive operations, the FPGA can often implement them in parallel. Therefore, they are capable of achieving high performance and can efficiently implement algorithms that manipulate bits and have parallelism. On the other hand, word-level operations

are expensive because of inefficient use of the logic cells and the limitations of internal routing. Modern FPGAs often introduce word-level processing elements to facilitate in word-level operations.

Application Specific Integrated Circuit

As its name implies, the ASIC is an IC designed for a specific application, or algorithm. This is in contrast with the GPP, DSP and FPGA which can be re-configured instead of configured only once. The non-reconfigurable nature of the ASIC enables synthesization at maximum performance. An advantage of the ASIC is therefore that, for any algorithm, the ASIC can beat the other programmable architectures by size, speed and energy-efficiency [11]. The downside is its inflexibility; for each algorithm, or update to an algorithm, a new chip has to be produced.

Conclusion

The trade-off among performance, flexibility and energy efficiency still is a decision based upon the target device. Mobile devices for example, require energy-efficiency on a sufficient level, whilst maximizing performance. Ideally the properties of the ASIC would be used, but since the ASIC can only perform one particular algorithm that is not always an option. Both the FPGA and the DSP are often too domain specific to be implemented in a general mobile device.

Reconfigurable architectures try to find the best compromise between these properties. They maintain their flexibility by being reconfigurable, whilst being energy-efficient and fast at the same time. Nevertheless, in the end a choice will always have to be made, because even different reconfigurable architectures have their own properties when it comes to specific algorithm domains [8].

3.2 Reconfigurable architectures

Reconfigurable architectures are defined as architectures with an adaptive instruction set. A subset of the instruction set can be defined as a "program" which can be executed by the hardware. Due to this reconfigurable nature, algorithms can be implemented using software, for running on these architectures. This saves costs and need for developing an algorithm-specific chip. Reconfigurable architectures have some distinct properties by which they can be identified and classified. Different algorithms require different approaches; as fully programmable architectures are inherently inefficient, each reconfigurable architecture targets a specific algorithm domain to achieve the desired trade-off.

Domain specific

Each algorithm is made up of different substructures. Fast Fourier transforms (FFTs) for example have a kernel that consists of an FFT butterfly, while the kernel of a Finite Impulse Response (FIR) filter consists of a Multiply Accumulate (MAC) operation. If the architecture is domain-specific it can

facilitate in such arithmetic by hardware. The more an architecture is domain specific, the more it can be optimized.

Reconfigurability

The frequency in which architectures have to switch configurations can range from mere seconds to once an hour or less. Infrequent reconfigurations are known as *static reconfiguration*, while frequent reconfigurations are known as *dynamic reconfiguration*. For dynamic reconfiguration it is required that the architecture can switch configurations quickly. Configuration data therefore has to be small. By sacrificing chip area, a second configurable space can be created which can be loaded in the background. The actual switching of configurations can then be performed instantly.

Granularity

Granularity is determined by the width of the components in its datapath. Heysters [11] considers an architecture to be fine-grained when the datapath is four bits wide of less. Having a width of more than four bits is considered to be a coarse-grained architecture. FPGAs are fine-grained because their logical units are optimized for bitwise operations. Fine-grained architectures therefore require a larger set of configuration data because of its "high resolution" of configurability.

Design automation

High-level design languages are of great importance when reducing the design-cycle. Low-level design tools such as HDLs or assembly, require a high level of knowledge of the underlying architecture. Exploiting parallelism and efficient routing is left to the programmer. High-level languages obscure the details of the architecture and rely on the compiler to extract and exploit parallelism of the architecture. Because compilers are nog always capable of delivering optimal solutions, the programmer still needs to direct the compiler. A solution is using a high-level language in which it is still possible to control the mapping of the algorithm, when desired.

3.3 Heterogeneous architectures

Heterogeneous architectures are architectures that are composed of different sub-architectures (e.g. a chip with a GPP and an other reconfigurable processor). Because different algorithms perform differently on various architectures, combining multiple architectures in one design has advantages. This could, if designed correctly, increase both performance and flexibility.

Mobile devices are used for a broad variety of applications. A mobile phone for example is capable of sending and receiving data on a cellular network. Furthermore it is capable of receiving and decoding video-streams, locally playing audio, and so on. The algorithms of these tasks vary from computational intensive to memory intensive ones. To accommodate for all of these in one architecture, whilst securing the performance requirements, is a challenge. A

dynamic reconfigurable architecture could perform all algorithms, ranging from the network data link level to the application layer [1].

The workload of mobile devices is ordinarily very diverse. Streaming and displaying video from a wireless network for example, is much more intensive than being in standby while only maintaining a network connection. Therefore, different parts of a heterogeneous architecture need to be put in low-power mode.

Heterogeneous architectures are architectures from which a set of sub-architectures can be enabled which is most suitable for the particular job. On low load multiple parts of the architecture can be switched off or put in low power mode. This aids in securing the performance and energy-efficiency requirements, while being very flexible.

3.4 Montium Architecture

In this thesis the Montium Tile Processor (TP) architecture is used [11]. This is done to provide a proof of concept for RS on this architecture. The Montium TP is a domain specific reconfigurable core (DSRC) that can be embedded in a coarse-grained reconfigurable processing tile.

The core of the Montium (Figure 3.3) contains five Arithmetic Logical Units (ALUs), capable of integer as well as fixed point calculations. Each ALU has its own register file containing four banks of four 16-bit values. Every ALU is connected to two memories via a local interconnect. A single memory has a capacity of e.g. 1024 16-bit values. The combination of the ALU, its registers and its two memories is known as a processing part (PP). The five processing parts are known as the processing parts array

The size of the memories, as well as the datapath width are parameterizable and can be configured on design-time. This thesis relies on the parameters of the "Annabelle" chip (section 3.4), for further explanation of the Montium. (PPA).

Since each register file consists of four banks, four values can be used simultaneously as input for the ALU (Figure 3.4). $n-1$ ALUs also have an "east" input, which is routed directly from the combinatorial part ("west" output) of the neighboring ALU.

The ALU also has two standard outputs. The internal part is divided into two levels. Level 1 consists of function units which are capable of executing standard logical, and simple arithmetical operations. Due to the way the registers are connected, not every input is available in every function unit. Level 2 of the ALU consists of a multiply-accumulate (MAC) followed by a butterfly structure.

Communication among PPs is provided by 10 global busses, which also allow for inter-PP communication. The busses can also be used to communicate with external components via the Communication and Configuration Unit (CCU). The CCU interfaces the Network on Chip (NoC). The CCU provides as many in- and output lanes to the Network on Chip (NoC) as is required by the network.

The Montium is controlled by its sequencer, which contains up to 256 40-bit instructions (Figure 3.5). Every instruction contains the directives for the ALU, registers, memories and interconnects. If an algorithm requires over

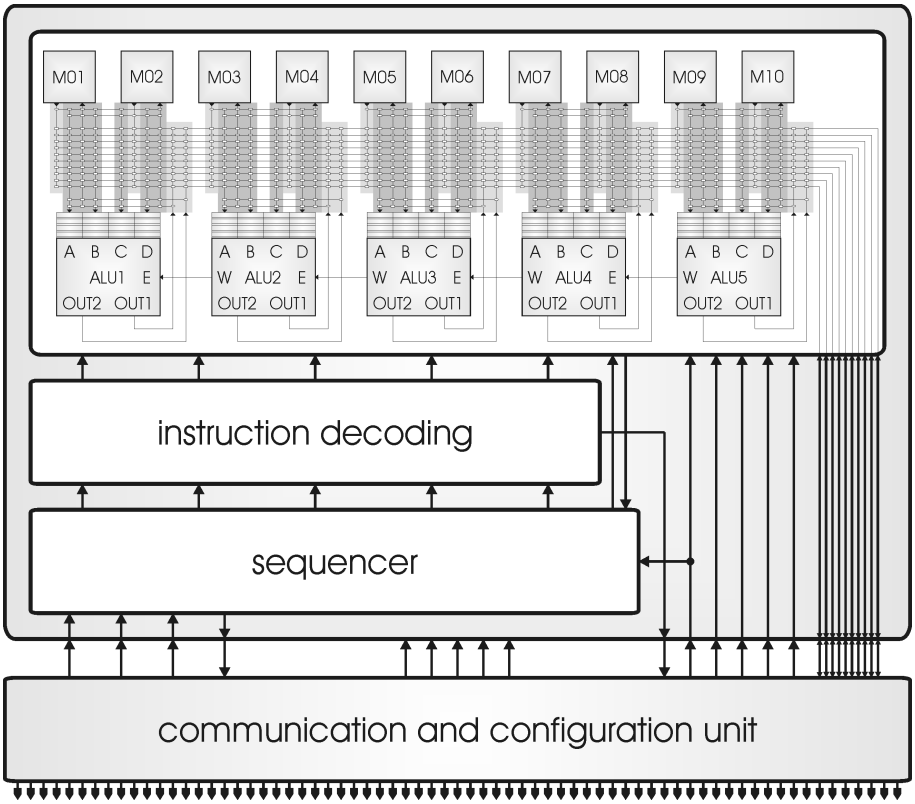


Figure 3.3: Structure of the MONTIUM TP [11]

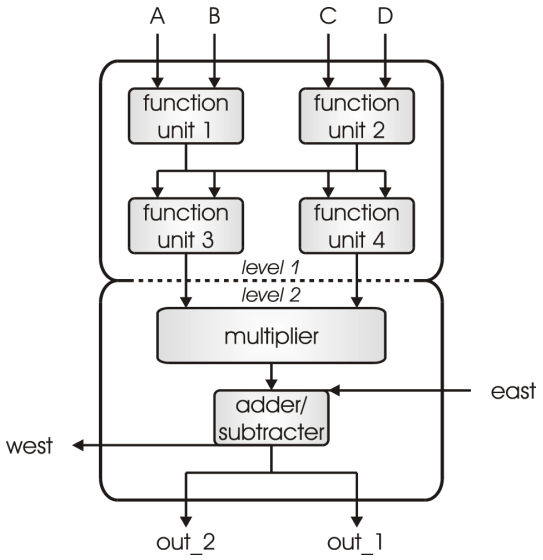


Figure 3.4: Structure of an ALU of the MONTIUM [11]

Figure 3.5: Sequencer instruction format [11]

256 instructions, the Montium is subject to reconfiguration. Through the use of sequencer-instructions the sequencer can perform conditional jumps, create loops, and enabling for self-modifying code.

The sequencer operates like a state machine, which decodes the states into hardware instructions in several stages. Due to the staged decoding of hardware instructions the configuration space and time is small. The sequencer instruction invokes a decoder instruction which selects the correct hardware instruction. The hardware instructions, as well as the decoder instructions, are stored in registers. Due to the limited capacity of these registers, not every combination of instructions is possible with a single configuration. This results in a subset of the possible instructions per configuration. Every ALU for example, has eight configuration registers, leading to a reconfiguration when more than eight are required.

Implementation

There are currently two Montium implementations available.

- FGPA Implementation

  The European 4s (Smart Chips for Smart Surroundings [5]) project is part of the Sixth EU's Framework Programme for Research and Technological Development. In this project, pushing the limits of CMOS was considered a priority. The Montium has been prototyped for this project, resulting in a BCVP (Basic Concept Verification Platform). In the BCVP the Montium was prototyped using a Xilinx XC2V8000 FPGA.

  This implementation of the Montium is limited to 6.6 MHz, worst-case path speed. This results in an input/output throughput of maximally 422 Mb/s.

- ASIC Implementation

  The "Annabelle" is an ASIC implementation of the Montium, developed by Recore Systems in cooperation with Atmel. The Annabelle is currently limited to 100 MHz. The input/ouput streams therefore have a throughput of 6.4 Gb/s. The type of CMOS used to create the ASIC is 130nm.

  These figures indicate that the Montium is suitable for algorithms with high computational and bandwidth requirements.

4

Encoder design and implementation

The RS encoder is capable of encoding data blocks. This encoding comes down to performing calculations which result in $n-k$ parity symbols. These symbols are then appended to the original message symbols to create a message of length n (Figure 2.2). The coefficients that are produced by the encoder ($P(X)$) are appended to the data ($D(X)$) to create $C(X)$. This section discusses the design and implementation of the RS encoder.

4.1 Structure

There are two different approaches that are commonly used to implement the functionality of an RS encoder.

Standard Encoding

Standard encoding can be used to calculate $P(X)$ by essentially using equation 2.9. This requires $n-(2t+1)+1 = n-(n-k) = k$ divisions between $D(X)$ and $G(X)$. $2t+1$ refers to the number of coefficients of the generator polynomial. A division in a Galois Field can be done by subtracting the values in the LUT of the field. This procedure can be described as dividing the first coefficient of D by the first coefficient of G, and subtracting that many times G of D. This is the very essence of a modulo operation. In short, the entire equation 2.9 procedure takes k individual Galois divisions, $(2t+1-1)\cdot k$ Galois multiplications and $(2t+1)\cdot k$ Galois subtractions.

Linear Feedback Shift Register

A common implementation is the use of the Linear Feedback Shift Register (LFSR), also referred to as an i-Stage Shift Register (SSR). The LFSR is a shift register in which the new state is a linear function of its input and previous state. The LFSR is commonly implemented in both hardware as well as software environments. The initial value of the LFSR is referred to as the "seed". The consecutive input symbols that are feeded to the LFSR are known as the "tap sequence".

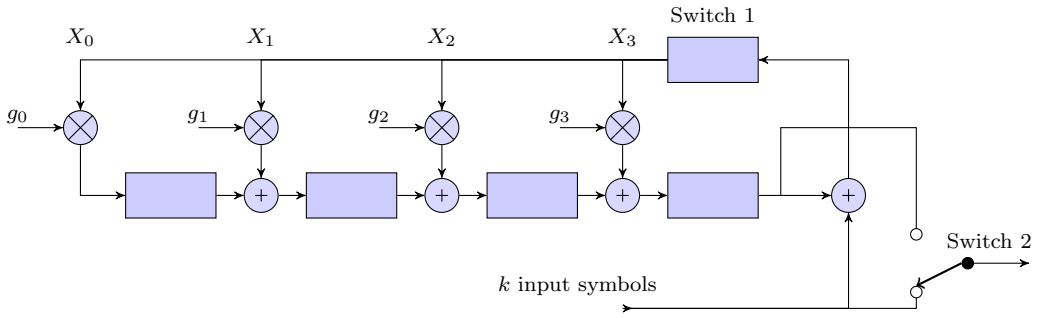


Figure 4.1: Structure of an $n - k = 4$ LFSR

In contrast to the ordinary LFSR, the Galois LFSR is not binary, yet comprises Galois symbols and arithmetic. The values of g^i represent the i^{th} coefficient of the generator polynomial $G(X)$. The α values represent the tap sequence i.e. the ordered message symbols. As the number of coefficients of $G(X)$ is $n - k = 2t$, the number of tabs is also $2t$. After processing the input ($D(X)$), the shift-registers contain the $2t$-sized $P(X)$. An example of an LFSR encoder for RS is given in figure 4.1. The structure is similar to the structure of a binary LFSR. The steps that are taken during the application of this LFSR can be described as follows:

1. Switch 1 is closed during the first k clockcycles in which the tap sequence is shifted in to the $(n - k)$-SSR.
2. Switch 2 connects the input directly to the output during the first k clockcycles, leading the input symbols directly to the output.
3. Once the k^{th} message symbol has passed the second switch, switch 1 is opened. Switch 2 then connects the output of the LFSR to the global output.
4. The remaining $(n - k)$ clockcycles shift the parity symbols in the shift-registers and moves them to the output. Because the input is zero the multiplication will result in the value zero, thus not adding anything while shifting the shift-register-values to the output.

The result of these operations is that the output of the LFSR contains the parity symbols ($P(X)$) and the message polynomial ($D(X)$). More formally specified as $P(X) + X^{n-k}D(X) = C(X)$.

Structure of choice

The structure of choice for RS encoding in this thesis is the LFSR. Designing and implementing the encoder was originally meant as an exploratory and introductionary try-out. Therefore it makes sense to try a different approach while designing. In this case the design is not based on a software- or mathematical model, yet a hardware structure. It can not be guaranteed that the solution itself is most efficient. It will be tried to achieve a simple one-on-one hardware-to-Montium mapping without special optimizations, which will

most likely introduce overhead. A single clockcycle in a hardware encoder could trigger numerous events at once, while the MONTIUM is only limited to five ALUs and limited memory-to-memory interaction per clockcycle. This would increase the number of required iterative steps, introducing predictable overhead. It is stressed that this design originates from a goal different than achieving the industries fastest and most optimal RS encoder.

Mapping and implementing a Reed-Solomon encoder on the MONTIUM has been done before in [14], based upon an iterative C implementation. Taking this under consideration yields an opportunity to test a different approach. Translating a hardware- instead of a software-construction on a MONTIUM-mapping has not been done for the encoder.

4.2 Design and implementation

The LFSR job can be split, like virtually any algorithm, into an input part, computation part and output part. This section will discuss the mapping of the LFSR to the MONTIUM target hardware.

LFSR steps

The steps that an LFSR takes in more detail, are derived from the general description in section 4.1.

1. During the first k clockcycles, the input symbols are streamed into the LFSR. The values are stored in the first computational unit which will add (XOR) the result of the previous LFSR state. At the same time, they are forwarded to the output.
2. The input symbol is added to the previous state of the LFSR, the result of which is copied to all inputs of the Multiply-accumulate (MAC) parts.
3. The MAC-parts will multiply the input with its generator coefficient. Such a Galois multiplication will be done by seeking the corresponding values for the generator coefficient and the input, in a LUT. The results are then added, and a reverse lookup takes place. The result is stored in a register.
4. After all MAC-parts have finished their computation, the shifting takes place. Every result takes the result of the previous MAC unit, and adds (XOR) that value to itself.
5. After the first k clockcycles there is no input left. Instead of moving the input values directly to the output, the LFSR is shifted empty. This is done by repeating step 4 $n - k$ times until all values have cleared.

These are the operations that need to be performed in order to calculate the RS message. To map this sequence of operations onto the MONTIUM requires at least assigning the memories, registers and PPs. There are almost an unlimited number of possibilities to map a sequence to a MONTIUM, but the final design will have to be optimized for performance and energy. To achieve performance parallelism needs to be exploited as much as possible. The design of the LFSR itself already provides support for parallelism; all MAC-units can be calculated in parallel.

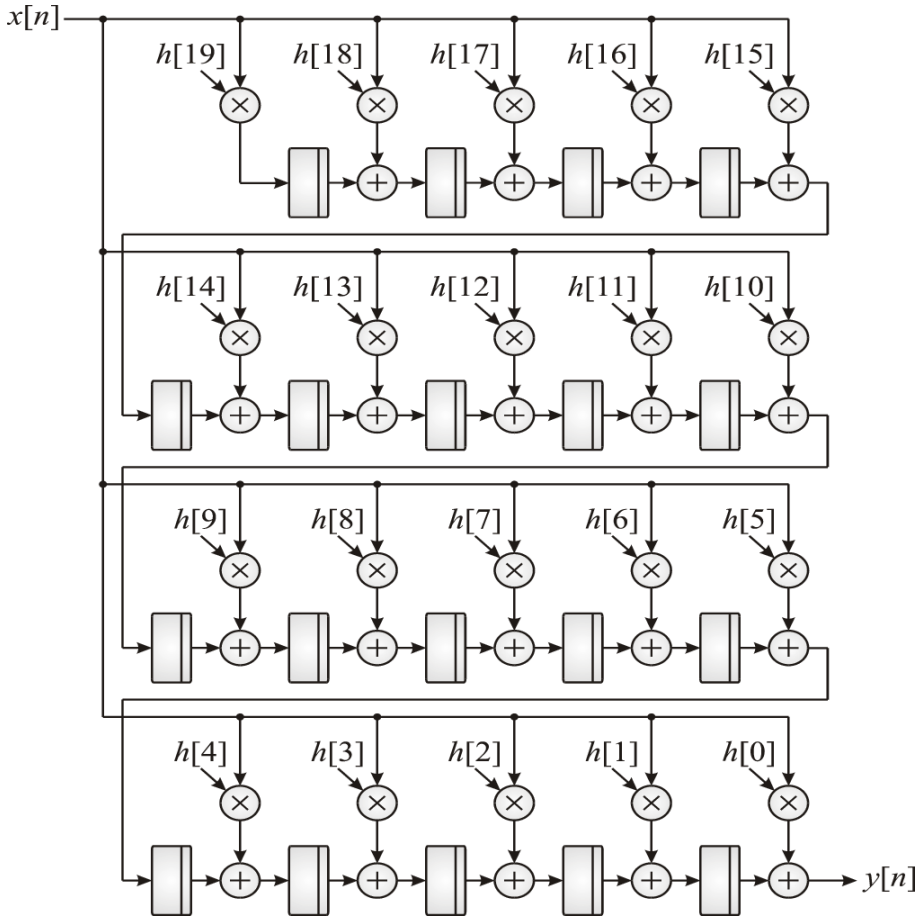


Figure 4.2: Example of a TDM design in a Montium [11]

Spatial distribution of ALU tasks

As there are only five ALUs, not every MAC-unit can be calculated at the same time. There are $n - k$ MAC-units meaning they have to be calculated in at least $\lceil((n-k)/5)\rceil$ subsequent clockcycles. The technique that can be used can be thought of as time division multiplexing (TDM). It resembles the technique described in chapter 6.3 of [11]. A large serial structure is then split up into smaller parts, which are calculated sequentially in time. When using ALU 1..5, the output value of ALU 5 can be feeded back to the input of ALU 1 in the next clockcycle, thus simulating serial behavior (Figure 4.2).

If the five ALUs are all used for the MAC operations, the other operations have to happen before or after. The only remaining operation is the input-state addition, which can be done by one ALU. To minimize configurations whilst not sacrificing flexibility, performance and energy-efficiency, an 4-ALU design is preferred. The remaining ALU can perform the input-state addition, while the other PPs can focus solely on MACs. The design choice can be justified by two means. Firstly the number of configurations is reduced. Because every

PP has its own specific task, there is no need for extensive contact switching. Every PP does need reconfiguration, otherwise it could only perform one single task every clockcycle. Because the MONTIUM supports partial reconfiguration this is not a costly operation. Secondly it can be noticed that the 4-ALU approach leaves one ALU idle most of the time. Research into the applications of RS codes has shown that most implementations use 2 or $4i, i \subseteq \{1..4\}$ parity symbols, with a maximum of 16. DVB also requires 16 parity symbols. A 4-ALU design therefore does not cause overhead when regarding clockcycles ($\lceil \frac{2t}{4} \rceil = \lceil \frac{2t}{5} \rceil, 2t = 2^x$). Together with the reconfiguration reflection it leads to the conclusion that the chosen design does not violate any of the primary requirements of the system.

As discussed the MONTIUM implementation requires a strategy for TDM. As there are up to 16 MACs and just four ALUs, the original LFSR-1-clockcycle-MAC-operations need to be split up. As the LFSR shifts from the left to the right, the most suitable approach is to divide the MACs as follows:

1. PP1..4 handle the four leftmost MACs
2. After that the last PP sends its result to the input of the first PP, to calculate the next 4 MACS
3. This continues untill all MACs have been calculated

Using this strategy enables the MONTIUM to divide serialized structures in to semi-parallel structures.

PP steps

PP 1..4 perform the MAC operations. These operations are almost identical for every MAC and ALU, with two exceptions. These exceptions concern the first and last of the MACs. In the first case the MAC does not contain an addition (section 4.1), because there is not previous neighbour to add data to. In the last case the output can not simply be past to the next MAC in line, but needs to be routed to different locations depending on whether the k input symbols have already been processed. This is actually not an ALU exception but a difference in the configuration of the crossbar. To minimize the amount of configurations, a special approach is introduced. The first MAC does not have a previous neighbour, but this can be faked by passing a zero into the addition-part of the MAC. This reduces the number of ALU reconfigurations because an ordinary identical MAC can then be used.

The steps PP 1..4 take can be schematically identified as:

1. Load the input value
2. Retrieve the log of this value
3. Retrieve the log of the generator coefficient g_i
4. Add the logs to perform a Galois multiplication
5. Retrieve the $\log^{-1}$ of the result
6. Add (XOR) the retrieved result to the just received result of the previous MAC

7. Send that result to the next PPs memory for temporary storage, or output if the PP is the last MAC

As the log-table operations require a large amount of resources, optimizations could seriously increase efficiency as well as performance. The following optimizations are therefore recognized and implemented:

1. The log value of the input of every MAC is the same during one LFSR-clockcycle. The input on the feedbackline is distributed to every MAC. Having every MAC retrieving the log of this value is therefore unnecessary. The log is retrieved immediately after the feedback is calculated, and stored in the register instead of its original value (which is never used anyway).

2. The log of every generator coefficient g_i is retrieved every clockcycle by every MAC. The actual value of the g_i is never used, but only the log value. Therefore this value can be retrieved once, and be used without any further lookups. As the Montium needs to be reconfigured if another type of RS is used, the g_i values have to be loaded on initialization. Instead of loading the original values, the log values could be loaded reducing overhead in the Montium. As the used code is known in advance, these small number of values can easily be precalculated before loading into the Montium.

The steps PP 5 takes are:

1. Read input

2. Add (XOR) input to result of the MACs to create feedback

3. Output the feedback to the log lookup table so the MACs can use it.

PP and memory configuration

This section discusses the exact memory- and register management which is used to facilitate the application and its implementation.

The PP 1..4 are all almost identical units regarding memory layout. The PPs need to be able to perform two separate tasks; the multiplication and addition. Both multiplication and addition require two parameters. The multiplication requires the log of the g_i and the log of the input, while the addition requires the result of the previous MAC and the result of its own multiplication. As the first PP functions as both the first MAC as well as intermediate MACs (every $1 + 4n, n \subseteq \{0..3\}$) it needs both the result of PP 4 as well as a 0 initially. A graphical representation can be found in figure 4.3.

PP 5 also requires two inputs; the result of ALU 4 when it is the last MAC, and the input (see Figure 4.4).

To be able to store the input, output and lookup tables, the memories are used. As every memory can only retrieve one value at a time, every log-lookup needs a separate memory. PP 1..4 all need a $\log^{-1}$ table. A log table is not required, due to the fact that the g_i coefficients are externally calculated and thus being log values in the Montium from the beginning. PP 5 does require a log table to lookup the log value of the feedback. Furthermore,

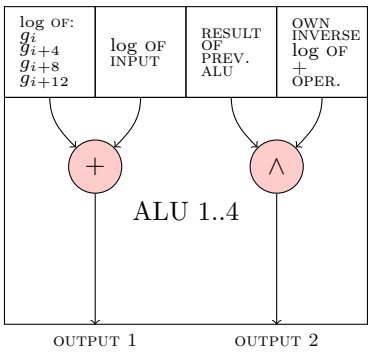


Figure 4.3: configuration of PP 1..4

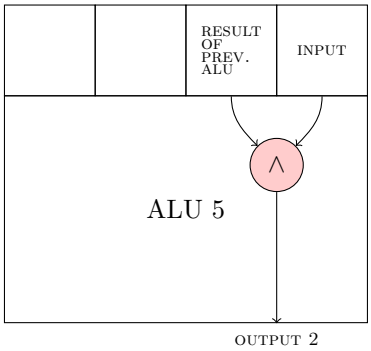


Figure 4.4: configuration of PP 5

a memory is used for the input and output to facilitate testing procedures without streaming capabilities. The memory-locations are labeled A..J with two consecutive memories belonging to one PP (i.e. PP1 has A and B, ...).

Implementation details

This subsection discusses two relevant implementation details, which may not be trivial. Firstly, the structure of the Montium code is discussed. Secondly, a note will be given explaining the problems and implementation of a one-clockcycle-RS-multiplication.

The Montium code consists of two nested loops. The outer loop loops k times, to process all input symbols. After some processing there is another loop which automates the TDM. Using loops like this stimulates the conservation of configurations and sequencer instructions.

RS multiplication

Performing an RS multiplication is less intuitive than it may appear. A multiplication is defined as follows, where A and B are the values to be multiplied:

```
if (A==0 || B==0) 0
```

#	Hex Value
0	0000
1	0000
2	0001
3	0019
4	0002
...	...

$\longmapsto$

#	Hex Value
0	0000
1	0400
2	0401
3	0419
4	0402
...	...

Figure 4.5: Modification to the log table for validity checking, incorrect implementation

```
else exp(log A + log B)
```

To verify if one of the original input values is zero or not, is complicated. The actual calculation (addition) is done using the log of the input values, not the original values. But $\log 0 = 0$ and $\log 1 = 0$, which causes some problems. In the first case, the algorithm should return zero, no matter what the second argument of the multiplication would be. In the second case the actual computation would be valid. It turns out there is no way to conclude from a log value whether its original value was zero or not, therefore making it impossible to decide whether to return zero or to do the actual calculation. As the input of the Montium ALU consists of the log values, an RS multiplication can not be performed.

A possible work around would require that based upon the log values, the ALU must be able to conclude whether the original value was zero. A solution has been found in the exploitation of a design choice in the Montium hardware. The memory interface uses 10bit for addressing lookup-values. As the interface uses only the LSBs of the addressing value, higher values can be used to form a validity-check. The first implementation of both the RS encoder and decoder, used a multiplication algorithm based on this technique, which was then proven to be incorrect. As the algorithm appeared to be valid, the next paragraph will explain its design and its shortcomings. After that, a correct algorithm is proposed.

Incorrect implementation We use the 11th bit of every log value to indicate that its original value was not zero. So, in the log table, 1024 is added to every value except the 0th (see figure 4.5). It can be seen that $\log 1$, although still seeming zero to the 10bit LSB addressed lookup table, can now be distinguished from $\log 0$ by the ALU.

This approach makes it possible to verify whether a value was originally zero or not. The Montium however, is not capable of performing the instruction

```
if(A >= 1024 && B >= 1024)
```

as it can only compare one value per clockcycle. This means the approach would only work if it can be guaranteed that A or B is never zero. This is generally not the case. To counter this problem the comparison has to be reduced to a single one.

Figure 4.7 shows the layout of the bits in one of the entries of the log table. The 10th bit (bit 9) is, as discussed, used for indicating the validity of the

#	Hex Value
0	0001
1	0002
2	0004
3	0008
4	0016
...	...

$\longmapsto$

#	Hex Value
0	0000
1	0002
2	0004
3	0008
4	0016
...	...

Figure 4.6: Modification to the exp table

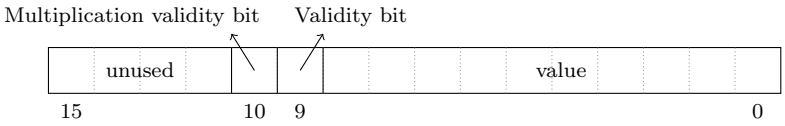


Figure 4.7: Layout of the bits in a 16-bit word, incorrect solution

A	B	$\log A$	$\log B$	$\log A + \log B$	AND 2048	Final exp(result)
0	0	0	0	0	0	0
0	1	0	1024	1024	0	0
1	1	1024	1024	2048	2048	0
1	2	1024	1025	2049	2048	2

Table 4.1: Calculation examples of the incorrect solution

value. The 11th bit is used for the validation of the multiplication. If both arguments were not originally zero, they will have the 10th bit set. If they are then added, the 11th bit will be set. If one of the arguments however was initially zero, the 11th bit will not be set. As the MONTIUM can only compare values to zero, masking can be used for the comparison. By performing an AND operation between the result and 2048 it can be concluded that if that result > 0, the multiplication is valid. Otherwise it is invalid. To make sure that the final output of the multiplication is zero, the *exp* table needs to be altered. The first entry is set to zero (see figure 4.6). This enables the ALU to output zero if the final result should also be zero. Remember that the first entry of the table is normally never used, as only $A + B = 0$ only is $A = 0$ and $B = 0$.

```
if( (A + B) AND 2048 ) > 0, then A + B, else 0
```

Unfortunately this algorithm is faulty for exactly one multiplication. Consider the case of the multiplication $1 \cdot 1$. This would translate into $\log 1 + \log 1 = 0x0400 + 0x0400$. The ALU notices that the addition is valid, because both 10th bits are set. Unfortunately the ALU will return zero, which makes the output exactly the same as the output of $0 \cdot 0$. Therefore, a distinction can not be made between $0 \cdot 0$ and $1 \cdot 1$ (see table 4.1), whereas the result should not be the same. The next paragraph proposes a solution to this problem.

A	B	$\log A$	$\log B$	$\log A + \log B$	ALU result	Final exp(result)
0	0	0	0	0	512	exp(512)=0
0	1	0	16384	16384	512	exp(512)=0
1	1	16384	16384	0 (with overflow)	0	exp(0)=1
1	2	16384	16385	1 (with overflow)	1	exp(1)=2

Table 4.2: Calculation examples of the correct solution

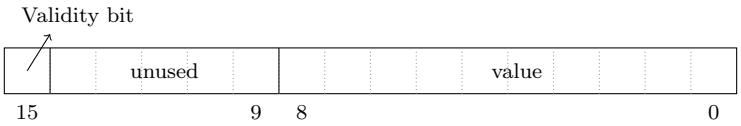


Figure 4.8: Layout of the bits in a 16-bit word, correct solution

Correct implementation The problem with the incorrect algorithm is that the addition-results 2048 and 0 will both lead to a final outcome of zero due to the AND. This makes it impossible to distinguish between a valid and an invalid multiplication. A solution is to use the overflow of the ALU. To facilitate the overflow, instead of the 11th bit, the MSB is set to 1 (value 16384) (see figure 4.8). The advantage is that after the addition, it is now possible to check for the overflow bit of the ALU. If it is set, the addition was valid. The difference to the previous incorrect solution is that here the AND is not required. As the Montium can only compare values to zero, it was not possible in the incorrect algorithm to check whether the resulting value was >= 2048, which would make sure that a distinction existed between $0 \cdot 0$ and $1 \cdot 1$. By using the overflow however, it is possible to perform such a check. The pseudocode is as follows:

```
if overflow (A + B), then (A + B), else 512
```

For this algorithm, the log table is altered to represent figure 4.9. The *exp* table is the original one, but with an appended value of zero on address 512 (see table 4.10). Table 4.2 shows the correct behavior of the algorithm.

The previous incorrect algorithm would work with the correct log and exp tables, if it was rewritten to:

```
if( (A + B) AND 2048 ) > 0, then A + B, else 512 //instead of 0
```

The Montium ALU however, is not able to perform the calculations and additionally pass the 512 value down from a register to the output. Generating the 0 internally is supported. Due to this limitation, the incorrect solution could not be fixed by replacing the "else" value.

#	Hex Value
0	0000
1	0000
2	0001
3	0019
4	0002
...	...

$\longmapsto$

#	Hex Value
0	0000
1	0800
2	0801
3	0819
4	0802
...	...

Figure 4.9: Modification to the log table for validity checking, correct implementation

#	Hex Value
0	0001
1	0002
2	0004
3	0008
4	0016
...	...

$\longmapsto$

#	Hex Value
0	0001
1	0002
2	0004
3	0008
4	0016
...	...
512	0000

Figure 4.10: Modification to the exp table

5 Decoder design and implementation

An RS decoder is capable of correcting errors by using the added parity symbols of the message. If no more than t errors are found, the decoder can correct the message to retrieve the original content. If there are more than t errors, the decoder can not reconstruct the original content. If the message has errors in such a way that the received message resembles another codeword, it could be corrected wrongly. In that case, the decoder outputs a corrected codeword without the ability to determine its validity. The decoder can be split up in several parts. The next section will discuss the structure of the decoder. The decoder is implemented in C [13], and conforms to the DMB/DVB standard $n = 204, k = 188$. Any absolute figures in this section are based upon $RS(204, 188), m = 8$.

5.1 Structure

An RS decoder can be split up in five parts (figure 5.1). Each part is connected through edges which resemble values or polynomials needed by the next part.

Syndrome calculator

The first part is the syndrome calculator (section 2.3). Syndrome calculation needs to be done for every $R(X)$ to determine its validity. The input of the syndrome calculator therefore is $R(X)$. The $2t$ syndromes are calculated by using Horner's scheme [16] or the check matrix [17]. Horner's scheme is selected because it evaluates a polynomial, whereas the check matrix relies on calculating the matrix and successively performing a matrix multiplication. If the syndrome polynomial $S(X)$ equals zero, then the decoding procedure is complete, and the message is considered valid. If one or more monomials $s_i \neq 0$ the RS decoder will start a search for the closest polynomial $R'(X)$ that does meet this criteria. Horner's scheme performs $4080(255 \cdot 16)$ Galois multiplications and $4080(255 \cdot 16)$ Galois additions per $R(X)$ (table 5.2).

Error Polynomial

This part of the decoder calculates the error locator polynomial $\Lambda(X)$. The Berlekamp-Massey algorithm is an improvement of the Berlekamp algorithm, and well studied. It is therefore considered to be an efficient algorithm for implementation [17]. Another available algorithm is Euclid's algorithm [16]. Unfortunately there is no algorithm which supports parallelism, so all calculation have to be done sequentially. The C code contains the Berlekamp-Massey algorithm. Using Berlekamp-Massey takes 592 additions and 848 multiplications.

Error Locations

The error locations can be found by localizing the roots of $\Lambda(X)$. As creating equations to solve this problem takes much more effort than brute forcing, the latter is preferred. This "Chien" search [17] calculates $\Lambda(X)$ for every $\alpha \in GF(2^m)$. A zero result indicates a root, thus an error location, has been found. As an RS code can only correct t errors, the output has a maximum of t symbols. Because of the independence of α the algorithm can be executed in 2^m parallel paths. The Chien search takes 4335 additions and 4335 multiplications.

Error Magnitude

The error magnitude block consists of two internal blocks. Forney's algorithm [15] describes a fast way to determine the error magnitudes based on the underlying mathematical structures. As his research has shown that the algorithm requires less calculations than the original Vandermonde matrix calculations [15] his algorithm is preferred. The first part is to calculate the error-evaluator polynomial $\Omega(X)$ (section 2.3). After that is done, Forney's algorithm can be used to derive the error values by equation 2.29. Constructing $\Omega(X)$ takes 2048 additions and 1024 multiplications. Forney's algorithm 392 additions and 392 multiplications.

Error Corrector

The error corrector corrects the actual $R(X)$ to $C(X)$. As its input consists of both the error magnitudes X and the error locations Y, the error magnitude X_i can be added to the Y_i^{th} monomial of $R(X)$ to correct that particular error. The output is the original reconstructed codeword $C(X)$. As there are only a maximum of t errors to correct, the error corrector uses up to 8 additions and 0 multiplications.

The polynomials that are calculated by each part have a predetermined maximum degree (table 5.1). Not all degrees are always required. For example, polynomial Y_i contains the error locations. If only two errors have occurred the polynomial will consist of just two monomials. As the forward communication among the blocks consists of these polynomials the size of this communication therefore depends on the actual number of errors.

To indicate the relative execution time of the different blocks, profiling information is available [9]. A DMB compliant RS implementation was profiled

Polyn.	Max. Degree
$R(X)$	$2t$
$S(X)$	$2t$
$\Lambda(X)$	$2t$
$\Omega(X)$	$2t$
X_i	t
Y_i	t

Table 5.1: Polynomials and their maximum degrees

Block	Additions	Multiplications	Parallel paths
Syndrome calculator	4080	4080	16
Error polynomial	592	848	1
Error locations	4335	4335	255
Error magnitude: evaluator	2048	1024	16
Error magnitude: errata	392	392	8
Error corrector	8	0	8

Table 5.2: Number of GF additions and multiplications per RS block [9]

Block	Time slice
Syndrome calculator	61.5 %
Error polynomial	6.01 %
Error locations	26.4 %
Error values	5.58 %
Error corrector	0.49 %

Table 5.3: Relative calculation time per RS block on an ARM7TDMI [9]

on an ARM (table 5.3), showing the relative execution time per block with a maximum number of errors. As the number of additions and multiplications influence these parameters, the results are as expected. Both the syndrome decoding and the error locator (Chien search) are found to be most time consuming.

5.2 Design and implementation

The goal of mapping RS to the Montium is to establish ideas on how to organize the internal structure of both the algorithm and the Montium to make full use of the efficiency and performance the Montium has to offer. As the Montium has five different PPs, making parallel processing one of its key features, the implemented algorithms should make full use of that. To be able to derive the proposed conclusions, two RS blocks have been selected and implemented on the Montium. The first block is the syndrome calculator. The syndrome calculator supports parallelism in 16 paths, and is computationally intensive. Furthermore, the syndrome calculator needs to be executed for every received $R(X)$. Having the Montium take care of this operation could therefore save a lot of execution time, thus optimizing overall through-

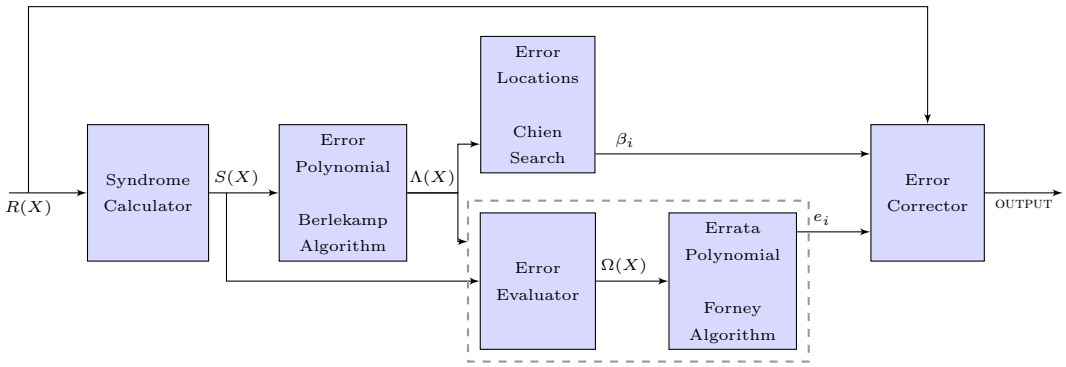


Figure 5.1: Computational parts of the RS decoder

put. The second implemented block is the Chien search which takes care of the error locations. This block is second in line concerning execution times, and is computationally the most intensive. Furthermore it supports a vast amount of parallel execution paths, making it ideal for Montium execution. If the Montium can efficiently take care of these two blocks, a combined 87.9% of the ARM execution-time would be taken care of. Depending on the efficiency and performance difference achieved by the Montium this might reduce overall execution time drastically, and thus increase the overall performance of the entire implementation.

While mapping, there are several constraints which have to be respected. Firstly, the inter operability between Montium and external processor. As the Montium itself is not capable of capturing RS data directly to memory, the data needs to be provided by an external source. Usually, an ARM is connected that can stream (configuration)data into the Montium. Every RS block can be implemented independently from the other blocks. This would mean that every block requires a reconfiguration of the Montium to load the algorithms. This would require the ARM to take care of the communication among the blocks, thus providing the system with a lot of communication overhead. A more ideal solution would be to have the ARM supply $R(X)$ just once to the Montium, and handling the entire decoding process in the Montium. The implementation will only support the syndrome calculator and the Chien search to be performed on the Montium. The design nevertheless needs to support the implementation of the other blocks as well. Therefore, the memories and ALUs need to be designed such that their configurations and data can be re-used by other blocks. The next section will discuss the mapping of the syndrome calculator, in which the memory design will be presented too.

Syndrome calculator

The syndrome calculator will in essence be an implementation of Horner's scheme. The major enhancement to the scheme will be the introduction of Galois arithmetic. This section will discuss Horner's scheme, mapping Horner's scheme to the Montium including Galois arithmetic, and general ideas on memory-design.

Horner's scheme

Horner's scheme is a method for evaluating a polynomial and its derivative at a constant value β. Note that we do not require, nor calculate the derivative. After the formal definition, an example will be given to illustrate how the algorithm actually works. Let $f(x)$ be a polynomial of degree m and β the constant to evaluate. Consider f_k to be the k^{th}-degree coefficient.

$$c_0 = f_m$$
$$c_i = f_{m-i} + \beta c_{i-1}$$

$$\text{thus } c_k = \sum_{i=m-k}^{m} \beta^{i-m+k} f_i$$
$$\text{And eventually } c_m = f(\beta)$$

As the formulae show, Horner's scheme consists of a recurrent relation that needs to be solved. To clarify the way in which the algorithm functions, an example is given. For clarification, the example is of Horner's scheme applied to standard mathematics. No fields are involved.

Example:

Consider the function $f(x) = 2x^4 + x^3 - x + 1$, and $\beta = -1$. Write the coefficients in descending order in a row:

f_4	f_3	f_2	f_1	f_0

In this case the coefficients are as follows:

2	1	0	-1	1

β should be written on the left side of the table. The highest coefficient f_m is copied as the first entry c_0 of the second row. Later entries shall be written as $c_i = f_i + \beta c_{i-1}$. The last entry is $f(\beta)$.

f_4	f_3	f_2	f_1	f_0
$c_0 = f_4$	$c_1 = f_3 + \beta c_0$	$c_2 = f_2 + \beta c_1$	$c_3 = f_1 + \beta c_2$	$c_4 = f_0 + \beta c_3$

The result of Horner's scheme with $\beta = -1$ is:

2	1	0	-1	1
2	-1	1	-2	3, *the result*

In this example, f_i translates to the i^{th} coefficient of R(X), and c_i is the recurrently calculated syndrome byte. The C implementation of Horner's scheme can be found in source code 5.1. For the mapping of Horner's scheme to the MONTIUM the structure of the scheme must be unfolded to a sequential or parallel list of operations. Stripping the code of unnecessary features results in the pseudo code of source code 5.1.

Sourcecode 5.1: syndrome calculator

```
1  //Function to calculate the syndrome of data[] using Horners'
       Scheme
2  get_syndrome(unsigned char data[], int nbytes)
3  {
4    //initialization variable
5    int i, j, sum;
6    int nMults = 0;
7    int nAdds = 0;
8    //NPAR is number of parity bytes: 16
9    //nbytes is a total of n bytes
10
11   //loop for every bit of data
12   for(i=0;i<nbytes; i++){
13     //loop for every syndrome byte
14     for(j=0;j<NPAR;j++){
15       //Horners' Scheme:
16       synBytes[j]=gadd( data[i], gmult(gexp[j+1], synBytes[j
             ]) );
17       //update statistics
18       nMults++;
19       nAdds++;
20     }
21   }
22 }
```

Sourcecode 5.2: Pseudo code of Horners' scheme

```
1  for i=0..255
2    for j=0..16
3      syndromebyte[j] = data[i] + ( gexp[j+1] * syndromebyte[j]
           );
```

Multiplication and addition

The design of adding multiplications, and performing the multiplications itself, is similar to the design used in the implementation of the encoder (section 4.2).

Design and re-usability

As the structure in source code 5.2 is used frequently in the RS decoder, its design must support re-usability to a certain extent. Apart from saving ALU configurations, a predefined way of mapping these structures will reduce the design cycle by offering a default approach. Furthermore, it assists in determining the best location of the log and exp tables. Studying the structure leads to the following conclusions:

- There are usually one or two loop counters that are used. They are only used for traversing arrays or indexing. Their actual value is mostly of no importance and is not referred to. There is an exception in the error locator, which can be dealt with in the design (see section 5.2).

- Multiplications: At most one of the arguments of the multiplications in the decoder must be calculated first. The other value is one that is already available, usually from memory.
- Addition: the result of the multiplication is always added to a value that is already available, usually from memory.
- The global structure is such that there is recurrence, which presents itself in two ways:

$$A = A + (B \cdot C) \tag{5.1}$$
$$C = A + (B \cdot C) \tag{5.2}$$

The order of execution is obvious as the result of the multiplication needs to be known before the addition can be performed. As one of the arguments of the multiplication may require pre-processing, this tops the execution order list:

$\mathcal{A}$. Calculate the multiplication's unknown argument

$\mathcal{B}$. Multiplication

$\mathcal{C}$. Addition

These three components will be referred to as $\mathcal{A}$, $\mathcal{B}$ $\mathcal{C}$, and "components". There are basically two different approaches that can be used for implementing this model on a Montium.

- **Data parallel approach** This means having every PP executing its own subset of data of the entire set. The drawback of that method is that every PP needs to reserve enough ALU configurations to suit the entire decoding model. Furthermore, it will not be possible to have five PPs execute the model in parallel, because too many memories (i.e. log, exp a.o.) have to be accessed simultaneously. Shifting the algorithm in time, by having each PP start its own iteration on a different clockcycle, could potentially solve part of this problem. Another drawback is that the addition ($\mathcal{C}$) requires the exp value of the result of the multiplication ($\mathcal{B}$). As this look-up takes one additional clockcycle, every PP's execution of every iteration of the algorithm will contain a one-clockcycle idle period for the ALU.
- **Functional parallelism approach** This approach is based on the structure of a waterfall. Every PP has its own designated task and passes its result on to another PP for further processing. The advantages of this method are that the number of ALU configurations is restricted, and that only a few instances of the lookup tables are required. The problem of the addition requiring the exp-result of the multiplication, is intrinsically solved. A drawback of the approach could be that other tasks that need to run in parallel to this algorithm, have less PPs at their disposal. However, the used RS decoder does not require any processing to take place whilst calculating syndromes.

Cycle↓	PP1	PP2	PP3	PP4	PP5
1	$\mathcal{A}1$				
2		$\mathcal{B}1$			
3			$\mathcal{C}1$		

Figure 5.2: Example of 'waterfall' model on PP1..5

Cycle↓	PP1	PP2	PP3	PP4	PP5
1	$\mathcal{A}1$				$\mathcal{A}'1$
2		$\mathcal{B}1$		$\mathcal{B}'1$	
3			$\mathcal{C}1$	$\mathcal{C}'1$	

Figure 5.3: Example of mirrored 'waterfall' model on PP1..5

The selected approach, is the multi PP approach. The execution-order list contains three different components. Each of these components can reasonably be performed in one clockcycle as far as the ALU is concerned. This means the waterfall model requires the use of at least three PPs (figure 5.2). Further optimizations are possible when considering the internal structure of the PP and the requirements of the algorithm. The addition is an XOR of two operands, both coming from memory. If designed correctly, a PP is capable of performing two of these additions every clockcycle, reducing the total mount of required PPs for the algorithm to 2.5 PPs. As the Montium contains five PPs it is possible to execute two instances of this algorithm in parallel, a "mirrored"-waterfall structure (figure 5.3).

Memories

The memories need to contain the log-table, exp-table and temporary results of the components. To prevent memories from being inefficiently accessed (e.g. located far way from the ALU), or unnecessarily duplicated, a generic design of the specific tables and results is required. The required tables per component are:

$\mathcal{A}$. The calculator, which calculates the input-value for the multiplication, does not always have the same requirements as to which tables are required. Researching the C implementation shows that only the exp-table is needed, and always in the following way:

```
gmult( gexp[A], B )
```

This shows that the presence of an exp table for $\mathcal{A}$ is required. $\mathcal{B}$ however, uses the log of both its arguments as its input. By realizing that:

$$\log(\exp(A)) = A, \tag{5.3}$$

the exp lookup becomes unnecessary. $\mathcal{A}$ therefore requires no specific tables of its own.

M1	INPUT DATA/NOT USED
M2	log
M3	Temporary sum
M4	Temporary sum
M5	exp
M6	exp
M7	Temporary sum
M8	Temporary sum
M9	log
M10	NOT USED

Table 5.4: Content of memories

$\mathcal{B}$. The multiplication requires the log of both arguments. Hence two log tables seem to be required, but as component $\mathcal{A}$'s output is already a log value, the number is reduced to one. Furthermore an exp table is needed to convert the result of the multiplication back to the original domain.

$\mathcal{C}$. The addition component requires no initial memories for its operation. However, due to the recurrence that occurs, two temporary storage facilities are required per $\mathcal{C}$ (See subsection 5.2).

It can be concluded that the mirrored-waterfall design requires a total of eight memories, four of which are used by $\mathcal{B}$ and $\mathcal{B}'$. Optimization can be achieved by looking more closely at the arguments of the multiplication. $\mathcal{A}$ usually already outputs the log of its calculation; looking it up is therefore not needed. There is one exception (Berlekamp Massey) in which the log of the syndrome bytes is required as input to the multiplication. For this particular instance a log table should be available. However, converting these 16 bytes to their log values in prior to executing the Berlekamp Massey algorithm that uses them, generates negligible overhead. The memories contain the content as specified in table 5.4. Their usage is explained in the next section.

Implementation

Horner's scheme iterates over $0..2^m - 1$ and within over $0..2t - 1$, and performs the syndrome calculation. As the model supports two independent syndrome bytes to be calculated in parallel, a choice has to be made as to how the loops are traversed. There are two possible options:

- **Divide the inner loop** Dividing the inner loop means looping $2^m - 1$ times, and performing the inner loop in two parallel paths.

```
                for(i=0;i<nbytes; i++){
for(j=0; j<NPAR/2; j++){ | for(j=NPAR/2; j<NPAR; j++){
  synBytes[j]=...        |   synBytes[j]=...
}                        | }
                         }
```

  E.g. the inner loop for $2t = 16$ is divided in 0..7 and 8..15.

- **Divide the outer loop** Dividing the outer loop means splitting the entire algorithm into two parts.

```
for(i=0;i<nbytes/2; i++){
  for(j=0; j<NPAR; j++){
    synBytes[j]=...
  }
}
```

```
for(i=nbytes/2;i<nbytes; i++){
for(j=0; j<NPAR; j++){
    synBytes[j]=...
  }
}
```

For the implementation the first construction is used. As the inner loop´s contents are in both cases the same, this is no argument to support any of the choices. However, the loop construction and its properties are influential. The advantage of the first construction is that the external data is processed one-by-one and in sequential order. The other construction would require two inputs to be available at the same time and in every clockcycle. To reduce the absolute difference in the index of the required input bytes, the algorithm could function in an "interleaved" way, by processing input-byte one and two, then three and four, etcetera. Due to the input-streaming advantages the first construction has, is has been selected for implementation.

The structure of the Montium mapping of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ for the syndrome calculator is given in figure 5.9. The actual design is discussed below. The clarification of why the components are not incrementally ordered from $\mathcal{A}1...\mathcal{A}8$, is given in the discussion of component $\mathcal{C}$ below. A detailed description of the components:

$\mathcal{A}$. The task of $\mathcal{A}$ is calculating the argument of component $\mathcal{B}$. The argument of the multiplication is exp`[j+1]`, which as previously discussed in section 5.2, results in the calculation of `j+1`. Due to splitting the inner loop, $\mathcal{A}$ needs to result in an output of 1..8 and $\mathcal{A}'$ in 9..16. Because of the way $\mathcal{C}$ requires the components to be ordered (see figure 5.9), the calculation is not a standard `j+1`. In the figure, $\mathcal{A}x$ needs to produce x on the output, while $\mathcal{A}'x$ needs to produce $x + 8$ on its output. Four different functions have to be supported:

- $(previousvalue + 1)$
- $(previousvalue - 1)$
- $(previousvalue - 4)$
- $(previousvalue + 7)$

For $\mathcal{A}'$, 8 needs to be added to this result. To save an ALU, and thus energy, both $\mathcal{A}$ and $\mathcal{A}'$ can be performed on one PP. The configuration of this PP look like figure 5.4.

$\mathcal{B}$. The multiplication requires two arguments. The first argument is the output of $\mathcal{A}$ which is received into the register. The second argument consists of an entry in one of the memories. In this case it is the j^{th} syndrome byte, that must be sent to the log table in advance. The multiplication is performed as previously discussed, and will therefore result in the Montium pseudo-code:

```
if overflow (registerA + registerB) then (registerA +
        registerB) else registerD -> output1
```

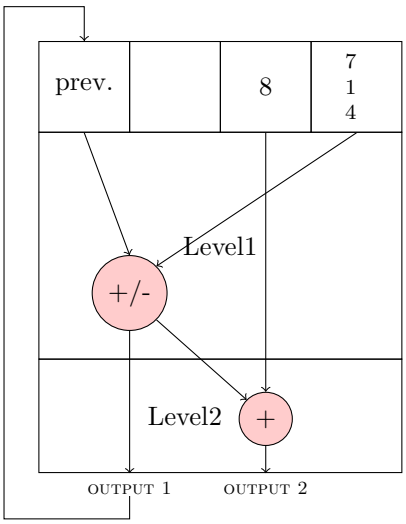


Figure 5.4: Configuration of PP for both $\mathcal{A}$ and $\mathcal{A}'$ on one single ALU

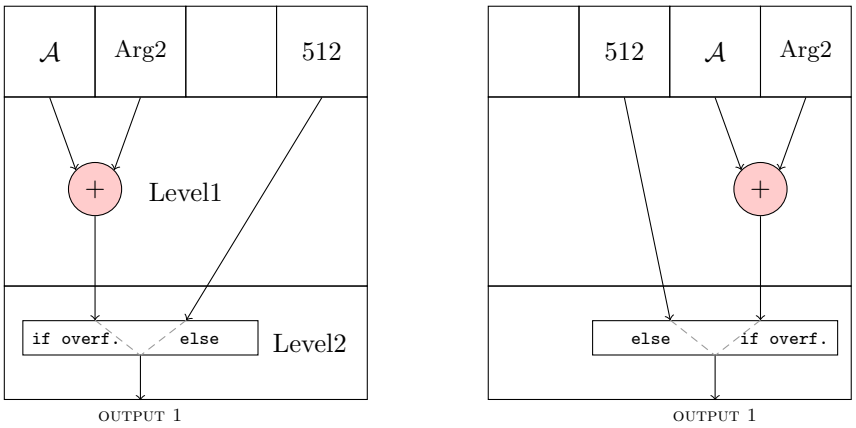
The next section on $\mathcal{C}$ will explain that there periodically is a value from $\mathcal{C}$ that will not be written to the SUM memories. This value will be temporarily stored in register D of $\mathcal{B}$. Because this value later needs to be used as input of the multiplication, the suggested configuration does not suffice. Therefore, another configuration will be introduced, which is in fact a mirrored version of the current configuration:

```
if overflow (registerC + registerD) then (registerC +
        registerD) else registerB -> output1
```

The multiplication can be performed in one clockcycle. The arguments have to be present in register A and B, or register C and D. Figure 5.5 shows the two configurations of the PP for this approach.

$\mathcal{C}$. The addition requires two arguments, and may use only half of a PP. The first input value is the exp value of the result of $\mathcal{B}$. Figure 5.8 shows the configuration of the PP for component $\mathcal{C}$. As the result is only available at the output of $\mathcal{B}$ in the clockcycle after its calculation, the look-up adds another clockcycle to the operation. This introduces a two-clockcycle delay for $\mathcal{C}$ with respect to $\mathcal{B}$ in the waterfall model. The other argument of $\mathcal{C}$ consists, in this case, of the previously calculated value of syndrome byte j. As there are 16 syndrome bytes, temporary storage in the registers is a problem. Memory M3, M4, M7 and M8 are reserved for this purpose. The process consists of these basic steps:

1. Load previous sum from memory
2. Perform addition
3. Store new sum back in memory



(a) Standard multiplication

(b) Periodical exception

Figure 5.5: Configuration of PP for $\mathcal{B}$ and $\mathcal{B}'$

The Montium can only write/read from one address per memory every clockcycle. If an address is specified, a write-operation to that address can be performed in the same clockcycle. A read-operation however, needs to be performed in the clockcycle after the address specification, as the value is only then available. The memories are designed to support read-after-write, meaning a value can be written as well as read in the same clockcycle. In that case, the newly written value will be the value that is read. This presents itself as a problem when wanting to use a single memory for both retrieval and storage of the temporary sums. To facilitate in both read and write operations every clockcycle, two memories have been reserved; one for writing and one for reading. While all $2t/2 = 8$ values are read from table A, their sums will be stored in table B. Therefore, every iteration both A and B need to switch with respect to read/write functionality. This will however introduce idle cycles in the process. Figure 5.6 illustrates the memory-interaction in combination with the components. The notation "$(\#x)$" will denote the x^{th} clockcycle of this figure. The "critical path" of one syndrome byte-iteration is six clockcycles. As $\mathcal{B}$ (#3) requires the log of the syndrome byte to be read, its address must be specified to the AGU two clockcycles ahead (#1). The specification of the write address, and the actual write, will occur in the clockcycle after $\mathcal{C}$, as its result is then available on the output of the PP (#6). This introduces a $\Delta 5$-clockcycle difference in the addressing of A and B, meaning that after the last value is read from table A (#1), it takes five clockcycles for its result to be available for writing into B (#6). After that operation, the memories need to switch functionality, and the first syndrome byte needs again to be read, but now from table B. Addressing the AGU for this can only happen in the clockcycle after that last write. It then takes an additional clockcycle for the first syndrome byte to be available for reading. This means that $\mathcal{B}1$ (#9) can only begin four clockcycles after $\mathcal{C}8$ (#5). In short, this will

Cycle↓	PPs	
1		Request value for $\mathcal{B}8$ from read-memory
2	$\mathcal{A}8$	Write value for $\mathcal{B}8$ as address into log table
3	$\mathcal{B}8$	Read value for $\mathcal{B}8$ from log table
4		
5	$\mathcal{C}8$	
6		Result of $\mathcal{C}8$ available on output of PP3 and written to memory
7		Request value for $\mathcal{B}1$ from write-memory (Which now is read-memory)
8	$\mathcal{A}1$	Write value for $\mathcal{B}1$ as address into log table
9	$\mathcal{B}1$	Read value for $\mathcal{B}1$ from log table
10		

Figure 5.6: Overview of memory interaction on read-write task-switch of memory

introduce an extra Δ5-clockcycle delay in the execution of each iteration (see figure 5.7). (Ideally $\mathcal{B}1$ would be executed in the clockcycle after $\mathcal{B}8$. Using this technique $\mathcal{B}1$ is delayed for five additional clockcycles). The efficiency of the algorithm will suffer, as every eight clockcycles, five idle cycles are required (62.5% overhead).

The solution to the problem is to use memory A and B for a disjunct group of the syndrome values. Memory A will store syndrome values 1..4 and B 5..8. The same will hold for $\mathcal{C}'$, for which memory $\mathcal{A}'$ will store 9..12 and $\mathcal{B}'$ 13..16. Using this approach "as is" will still introduce an idle cycle every four clockcycles, because switching the read/write functionality takes one additional clockcycle. This would still mean an overhead of 25%. Solving this problem requires not storing/reading all values in memory, thereby creating an idle cycle in the process in which the memories can switch. Figure 5.9 shows the mapping of the syndrome calculator. On the right, the addressing of the sum memories for either $\mathcal{C}$ of $\mathcal{C}'$ is shown. As can be seen, the left memory only handles syndrome bytes 1..4 and the right memory 5..8. Consider clockcycle 8. Instead of writing the result of $\mathcal{C}4$ of cycle 7 to memory, it is directly written to a log table. A clockcycle later, the log value is available, which is then stored in register D of $\mathcal{B}$. In clockcycle 10, this log value is used by $\mathcal{B}4$. As has been explained in the discussion about $\mathcal{B}$ above, this is where $\mathcal{B}$ uses the mirrored version of its original configuration. Hereby being able to process the just stored syndrome byte 4. This reduces the idle cycles from 25% to 0%, making optimal use of the Montium.

To save sequencer space, the actual implementation contains loops to enable iterative execution without specifying each iteration on its own. The "intro" of the construction (cycles 0 to 3) shows dissimilarities from further cycles, in the sense that there is no intermediate execution on PP2 and PP3. The "outtro" of the construction (25 till end) has no need to invoke another iterative step, and is therefore only required to finish up. To make the best use of a sequencer

Cycle↓	PP1	PP2	PP3	PP4	PP5
9	$\mathcal{A}8$	$\mathcal{B}7$	$\mathcal{C}5$ $\mathcal{C}'5$	$\mathcal{B}'7$	$\mathcal{A}'8$
10		$\mathcal{B}8$	$\mathcal{C}6$ $\mathcal{C}'6$	$\mathcal{B}'8$	
11			$\mathcal{C}7$ $\mathcal{C}'7$		
12			$\mathcal{C}8$ $\mathcal{C}'8$		
13					
14					
15	$\mathcal{A}1$				$\mathcal{A}'1$
16	$\mathcal{A}2$	$\mathcal{B}1$		$\mathcal{B}'1$	$\mathcal{A}'2$
17	$\mathcal{A}3$	$\mathcal{B}'2$		$\mathcal{B}'2$	$\mathcal{A}'3$
18	$\mathcal{A}4$	$\mathcal{B}3$	$\mathcal{C}1$ $\mathcal{C}'1$	$\mathcal{B}'3$	$\mathcal{A}'4$

Figure 5.7: A five clockcycle delay per memory-switch

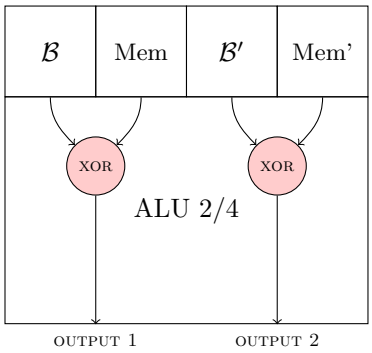


Figure 5.8: Configuration of PP for $\mathcal{C}$

loop, an intermediate part has been designated for looping (cycle 9 to 24). Repeating this part $\frac{n-2}{2}$ times is similar to expanding the figure and drawing the part $\frac{n-2}{2}$ times. The $\mathcal{A}'$ can be combined with $\mathcal{A}$ as discussed. The actual implementation implements this design to prove that it is possible to cope with four ALUs. Section 5.2 will discuss an example of a five-ALU-design.

Design and mapping of Error Locator

The error locator will in practice be an implementation of a Chien search. The major difference with respect to the regular version of Chien's algorithm is the application of Galois arithmetic instead of regular arithmetic. This section will discuss Chien's algorithm and its mapping to the Montium. Chien's algorithm is designed for finding the roots of a polynomial. Chien proposes an intuitive way, which resembles a brute-force approach. The basic idea is to evaluate the polynomial for all possible values of the used mathematical field. All results of zero are roots of the evaluated polynomial. Consider Λ to be the error locator polynomial, λ_j its j^{th} coefficient and r the value to evaluate Λ for. Evaluating

Loop $\frac{n-2}{2}$ times

Cycle↓	PP1	PP2	PP3	PP4	PP5	Read	Write	To log
0						1		
1	$\mathcal{A}1$				$\mathcal{A}'1$	2		
2	$\mathcal{A}2$	$\mathcal{B}1$		$\mathcal{B}'1$	$\mathcal{A}'2$	3		
3	$\mathcal{A}3$	$\mathcal{B}2$		$\mathcal{B}'2$	$\mathcal{A}'3$	4		
4	$\mathcal{A}4$	$\mathcal{B}3$	$\mathcal{C}1$ $\mathcal{C}'1$	$\mathcal{B}'3$	$\mathcal{A}'4$	Write	5	
5	$\mathcal{A}5$	$\mathcal{B}4$	$\mathcal{C}2$ $\mathcal{C}'2$	$\mathcal{B}'4$	$\mathcal{A}'5$	1	6	
6	$\mathcal{A}6$	$\mathcal{B}5$	$\mathcal{C}3$ $\mathcal{C}'3$	$\mathcal{B}'5$	$\mathcal{A}'6$	2	7	
7	$\mathcal{A}7$	$\mathcal{B}6$	$\mathcal{C}4$ $\mathcal{C}'4$	$\mathcal{B}'6$	$\mathcal{A}'7$	3	8	
8	$\mathcal{A}8$	$\mathcal{B}7$	$\mathcal{C}5$ $\mathcal{C}'5$	$\mathcal{B}'7$	$\mathcal{A}'8$	Read	Write	4
9	$\mathcal{A}4$	$\mathcal{B}8$	$\mathcal{C}6$ $\mathcal{C}'6$	$\mathcal{B}'8$	$\mathcal{A}'4$	3	5	
10	$\mathcal{A}3$	$\mathcal{B}4$	$\mathcal{C}7$ $\mathcal{C}'7$	$\mathcal{B}'4$	$\mathcal{A}'3$	2	6	
11	$\mathcal{A}2$	$\mathcal{B}3$	$\mathcal{C}8$ $\mathcal{C}'8$	$\mathcal{B}'3$	$\mathcal{A}'2$	1	7	
12	$\mathcal{A}1$	$\mathcal{B}2$	$\mathcal{C}4$ $\mathcal{C}'4$	$\mathcal{B}'2$	$\mathcal{A}'1$	Write	Read	8
13	$\mathcal{A}8$	$\mathcal{B}1$	$\mathcal{C}3$ $\mathcal{C}'3$	$\mathcal{B}'1$	$\mathcal{A}'8$	4	7	
14	$\mathcal{A}7$	$\mathcal{B}8$	$\mathcal{C}2$ $\mathcal{C}'2$	$\mathcal{B}'8$	$\mathcal{A}'7$	3	6	
15	$\mathcal{A}6$	$\mathcal{B}7$	$\mathcal{C}1$ $\mathcal{C}'1$	$\mathcal{B}'7$	$\mathcal{A}'6$	2	5	
16	$\mathcal{A}5$	$\mathcal{B}6$	$\mathcal{C}8$ $\mathcal{C}'8$	$\mathcal{B}'6$	$\mathcal{A}'5$	Read	Write	1
17	$\mathcal{A}1$	$\mathcal{B}5$	$\mathcal{C}7$ $\mathcal{C}'7$	$\mathcal{B}'5$	$\mathcal{A}'1$	2	8	
18	$\mathcal{A}2$	$\mathcal{B}1$	$\mathcal{C}6$ $\mathcal{C}'6$	$\mathcal{B}'1$	$\mathcal{A}'2$	3	7	
19	$\mathcal{A}3$	$\mathcal{B}2$	$\mathcal{C}5$ $\mathcal{C}'5$	$\mathcal{B}'2$	$\mathcal{A}'3$	4	6	
20	$\mathcal{A}4$	$\mathcal{B}3$	$\mathcal{C}1$ $\mathcal{C}'1$	$\mathcal{B}'3$	$\mathcal{A}'4$	Write	Read	5
21	$\mathcal{A}5$	$\mathcal{B}4$	$\mathcal{C}2$ $\mathcal{C}'2$	$\mathcal{B}'4$	$\mathcal{A}'5$	1	6	
22	$\mathcal{A}6$	$\mathcal{B}5$	$\mathcal{C}3$ $\mathcal{C}'3$	$\mathcal{B}'5$	$\mathcal{A}'6$	2	7	
23	$\mathcal{A}7$	$\mathcal{B}6$	$\mathcal{C}4$ $\mathcal{C}'4$	$\mathcal{B}'6$	$\mathcal{A}'7$	3	8	
24	$\mathcal{A}8$	$\mathcal{B}7$	$\mathcal{C}5$ $\mathcal{C}'5$	$\mathcal{B}'7$	$\mathcal{A}'8$	Read	Write	4
25	$\mathcal{A}4$	$\mathcal{B}8$	$\mathcal{C}6$ $\mathcal{C}'6$	$\mathcal{B}'8$	$\mathcal{A}'4$	3	5	
26	$\mathcal{A}3$	$\mathcal{B}4$	$\mathcal{C}7$ $\mathcal{C}'7$	$\mathcal{B}'4$	$\mathcal{A}'3$	2	6	
27	$\mathcal{A}2$	$\mathcal{B}3$	$\mathcal{C}8$ $\mathcal{C}'8$	$\mathcal{B}'3$	$\mathcal{A}'2$	1	7	
28	$\mathcal{A}1$	$\mathcal{B}2$	$\mathcal{C}4$ $\mathcal{C}'4$	$\mathcal{B}'2$	$\mathcal{A}'1$	Write	Read	8
29	$\mathcal{A}8$	$\mathcal{B}1$	$\mathcal{C}3$ $\mathcal{C}'3$	$\mathcal{B}'1$	$\mathcal{A}'8$	4	7	
30	$\mathcal{A}7$	$\mathcal{B}8$	$\mathcal{C}2$ $\mathcal{C}'2$	$\mathcal{B}'8$	$\mathcal{A}'7$	3	6	
31	$\mathcal{A}6$	$\mathcal{B}7$	$\mathcal{C}1$ $\mathcal{C}'1$	$\mathcal{B}'7$	$\mathcal{A}'6$	2	5	
32	$\mathcal{A}5$	$\mathcal{B}6$	$\mathcal{C}8$ $\mathcal{C}'8$	$\mathcal{B}'6$	$\mathcal{A}'5$	Read	Write	1
33		$\mathcal{B}5$	$\mathcal{C}7$ $\mathcal{C}'7$	$\mathcal{B}'5$			8	
34			$\mathcal{C}6$ $\mathcal{C}'6$				7	
35			$\mathcal{C}5$ $\mathcal{C}'5$				6	
36							5	

Addressing of sum-memories per clockcyle for $\mathcal{B}$

Figure 5.9: Mapping of the elements of the syndrome calculator

the polynomial once can be done using the following equation:

$$\Lambda(r) = \sum_{j=1}^{2t} (\lambda_j r^j) + 1 \tag{5.4}$$

A polynomial representation f_{chien} of the result of the entire Chien algorithm can also be mathematically derived:

$$f_{chien}(X) = \sum_{r=1}^{n} \left(\sum_{j=1}^{2t} \lambda_j r^j \right) X^{r-1} \tag{5.5}$$

The Λ polynomial has $2t+1$ λ coefficients. In practice however, it contains only as much successive non-zero λ_j values as there are errors in $R(X)$, thus $\lambda_j, j \in 1..v$. This has a minimum of 1 (otherwise calculating Λ would be useless), and a maximum of t. If it contains more than t errors, the $R(X)$ would be uncorrectable, making the Chien search unnecessary. This means the equation can be rewritten to:

$$\Lambda(r) = \sum_{j=1}^{v} (\lambda_j r^j) + 1 \tag{5.6}$$

The C implementation of the Chien search can be found in source code 5.3. For the mapping to the Montium the structure of the algorithm must be unfolded to a sequential or parallel list of operations. Converting it to pseudo code leaves source code 5.4, where $NPAR$ denotes the number of parity symbols. Note that in this research the worst case of $v = t$ will be discussed.

There is a difference between source code 5.3 and 5.4. As can be seen in equation 5.4, the λ_0 is known a priori to be 1. This can be taken advantage of in the first iteration of the algorithm, see line 14 of 5.3. As ktemp is zero, the `gexp[(ktemp*r)%255]` will always result in 1. `Lambda[ktemp]` will also result in 1. As the multiplication will then also result in 1, the sum will be always 1 after the first iteration. By initializing the sum to 1, and limiting the iteration over ktemp to 1..8 instead of 0..8, an iteration can be saved.

Sourcecode 5.3: Chien search

```
/* Finds all the roots of an error-locator polynomial with
    coefficients
 * Lambda[j] by evaluating Lambda at successive values of r.
 * This algorithm is worst case in the sense that is
    evaluates for t+1 coefficients instead of v+1
 */
void Find_Roots (void){
  int sum, r, ktemp;
  NErrors = 0;
  int nMults = 0;
  int nAdds = 0;

  for (r = 1; r < 256; r++) {
    sum = 0;
    /* evaluate lambda at r */
    for (ktemp = 0; ktemp < t+1; ktemp++){
      sum = gadd( sum, gmult( gexp[(ktemp*r)%255], Lambda[
          ktemp]) );
    }
  }
}
```

Sourcecode 5.4: Chien search pseudo code

```
//This algorithm is worst case in the sense that is evaluates
    for t coefficients instead of v
for r=1..255
  sum = 1
  for ktemp=1..t
    sum = sum + ( gexp[(ktemp*r)\%255] * Lambda[ktemp]))
```

The design of the syndrome calculator, as explained before, has already taken the global structure of this algorithm into account. Despite the similarities of the MAC operation, there are two key differences.

- Whereas the syndrome calculator performs equation 5.2, the Chien's algorithm consists of repeated adding to the same variable (equation 5.1). This means a slightly different approach should be chosen than used for the syndrome calculator.
- The calculation of the argument for the multiplication is less trivial than it is for the syndrome calculator. The modulo operation, with a value that is not expressible as a power of 2, means a dedicated ALU is required.

As the syndrome calculator implementation already has taken all MAC operations into account in its design, very little needs to be changed. The memories shall be kept the same (Table 5.4), including the global structure of the mapping (waterfall).

The Chien search iterates $1..2^m - 1$ times over $1..t$ and evaluates the error locator polynomial for every value 1..255. The model supports two independent evaluations to take place at the same time. It will also support the independent evaluation of the left and right part of the polynomial. In that case however,

the results of both have to be added to obtain the final result. The two constructions are:

- **Divide the inner loop** Dividing the inner loop means looping $2^m - 1$ times, and performing the inner loop in two parallel paths.

```
                                  for(r=1; r<256; r++){
for(ktemp=1; ktemp<v/2+1; ktemp++){ | for(ktemp=t/2+1; ktemp<v+1; ktemp++){
  sum = ...                         |   sum=...
}                                   | }
                                  sum=sumleft+sumright
                                  }
```

  E.g. the inner loop for $t = 8$ is divided in 1..4 and 5..8.

- **Divide the outer loop** Dividing the outer loop means splitting the entire algorithm into two parts.

```
for(r=1; r<256/2; r++){             | for(r=256/2; r<256; r++){
  for(ktemp=1; ktemp<v+1; ktemp++){ | for(ktemp=1; ktemp<v+1; ktemp++){
    sum=...                         |     sum=...
  }                                 |   }
}                                   | }
```

The first option introduces overhead in the form of an extra addition of the result. Another problem presents itself in calculating the initial value of $\mathcal{A}$ for the rightmost loop, which is discussed in the detailed description of $\mathcal{A}$ (see below). Furthermore, if v is uneven the amount of iterations on the left and right would not be the same. An advantage of the second option is that both the left and right part require the same Λ_j value at the same time, making reading it from memory less complicated. The problem of an uneven v is of no significance in this option. This second option has therefore been selected for implementation.

The design of the application for the Montium implements the worst case scenario of having $v = 8$ errors. Section 5.4 will discuss the case of a variable v.

The components $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ serve the same purpose and have the same layout as for the syndrome calculator. In fact, the design was originally made to fit multiple algorithms including the Chien search. The same preconditions still apply, due to the loading of parameters which is caused by log conversions in memory. The structure of the components $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ for the Chien search is given in figure 5.13. The actual design and clarification of the idle cycles is explained below, in the detailed description of the components:

$\mathcal{A}$. The task of $\mathcal{A}$ is calculating the argument of component $\mathcal{B}$. The argument of the multiplication is `exp[(ktemp*r)%255]` which, as discussed in the section of the syndrome calculator, results in `(ktemp*r)%255`. A regular one-clockcyle calculation of this result is not possible on the Montium. `ktemp*r` can be calculated, but a multiplication is only possible in level 2 of the ALU. A solution for the modulo operator is then beyond the capabilities of the hardware. Optimization of the structure of this calculation can be achieved by realizing how the different components of the calculation are related. Every execution of $\mathcal{A}$, `ktemp` increases by 1. `r` will only

increase once for every calculated sum (i.e. iteration). This can therefore be set on the "initialization" of every sum calculation. `ktemp` is the only value that changes during the execution of one sum calculation. It is also known in advance that `ktemp` increases by 1 every time $\mathcal{A}$ is executed. Therefore `ktemp*r` does not need to be calculated as a multiplication. It can be rewritten to an addition with a recurrent relation, where β_i is the result of the sum the i^{th} execution of $\mathcal{A}$:

$$\begin{aligned} \beta_0 &= r_{previous} + 1 \\ \beta_n &= \beta_{n-1} + r, n \in \{1..t-1\} \end{aligned} \tag{5.7}$$

Because the first calculation of `ktemp*r` needs to be calculated for `ktemp=1`, the first result of $\mathcal{A}$ needs to be r. Therefore, the first execution of a new loop will output $r_{previous} + 1$.

If 255 would have been expressible as a power of two, the problem of the modulo operation could be solved by bit masking. As this is not the case, a different approach must be taken. Every execution of $\mathcal{A}$ performs an addition, and must be able to take the modulo of this addition in the same clock cycle. Good use can be made of the observation that r ranges from 1..255. If any r is added to a certain correct value of β, the result might be out of the field (thus > 255), but never over 510 ($2 \cdot 255$). Therefore the modulo operation can be replaced by a conditional subtraction. If $\beta_{new} < 255$ nothing needs to happen. If however $\beta_{new} \geq 255$, a subtraction by 255 will be sufficient. As the MONTIUM is not able to determine if a value is above a certain non-zero value, the entire computation needs to be rewritten to a form in which the comparison can be done by zero. The final solution is to determine if $\beta_{new} - 255 \geq 0$. If so, β_{new} was originally out of its original field, and β_{result} is $\beta_{new} - 255$. Else β_{result} is β_{new}.

Summary: On the first execution of every loop of the sum-calculation, r will be increased by one($r_{init} = r_{previous\_loop} + 1$). This results in the correct output for the first iteration. Both the register that contains β, and the register that contain r, will in the second clockcycle be set to that value. Every next execution of $\mathcal{A}$ the regular computation will be performed, and sent to the output. The output can then be read by $\mathcal{B}$, but will also be lead back to $\mathcal{A}$ as it is required for the recurrent calculation. Figure 5.10 shows the configurations of the PP for $\mathcal{A}$.

This approach creates a problem with respect to the multiplication. As every argument of the multiplication needs to have the 16th bit set for validation, this solution does not suffice. There is however no solution possible that is also capable of having the 16th bit set appropriately. In this case, the multiplication has to be redesigned in order to support the absence of this bit.

$\mathcal{B}$. The multiplication is different from the one in the syndrome decoder, due to the reason given in the discussion of $\mathcal{A}$ above. The multiplication has two arguments; the output of $\mathcal{A}$ (argument A) and the $\log(\lambda)$ (argument B). Because $\mathcal{A}$ is not capable of producing the validation bit, another

method needs to be implemented. In this case a solution has been found in the "static" Λ array. If it can be guaranteed that there is no 0 in this array, only the input from $\mathcal{A}$ has to be checked. To ensure that there is no 0 in the Λ array, all zeroes need to be set to 256 before executing the Chien search. At the log table, address 256 is set to 511. Address 256 is originally not used, and therefore available. The log value of 256 will then be 511, resulting in a log vale of 511 for every Λ value that was zero. The Montium then executes:

```
if A equals 0, then -1, else A+B
```

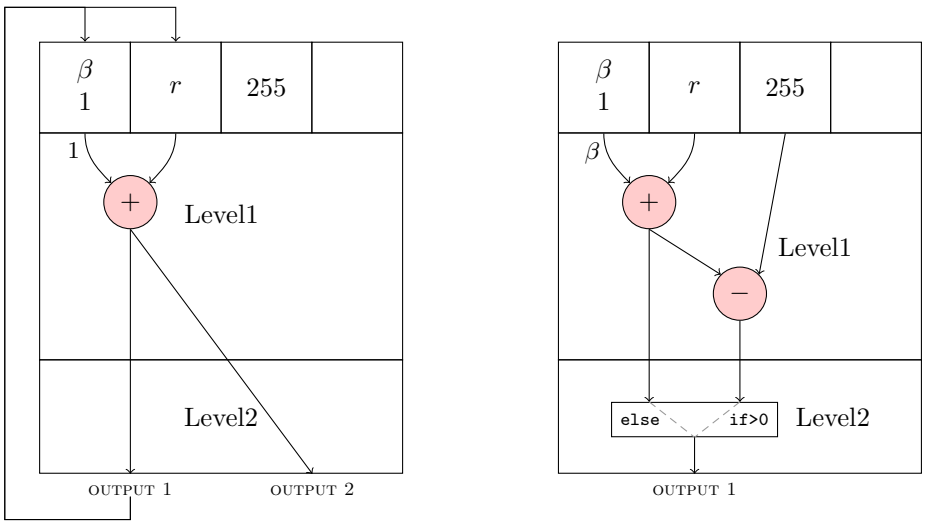
If A is zero, -1 is outputted. This can be generated internally in the ALU as logic_true. This results in a lookup of address 1023 in the exp table, which must be set to 0. If A is not zero, a normal addition is performed. If the Λ was originally zero, B is set to 512. As the exp table has length 511, any addition to 512 will result in a value $512 \leq result \leq 766(511 + 255)$. The exp table must therefore be extended to address 766 with zeros. The memory can not be used for other purposes than the exp table, because that table is used every clockcycle. It is therefore not a problem to allocate more of this memory to the exp table. Figure 5.11 shows the ALU configuration for $\mathcal{B}$.

$\mathcal{C}$. The addition is almost the same as for the syndrome calculator. Internally, the design remains the same. However, due to the structure of equation 5.1 the sum of the previous clock cycle serves as input for the new sum. Therefore a feedback from the output to the input register needs to exist (Figure 5.12). On initialization of every iteration, the sum needs to be set to 1. To realise this, the first iteration of every loop not the regular sum value will be read, but a different register that contains 1.

To save sequencer space, the actual implementation contains loops to enable iterative execution without specifying each iteration itself. The "intro" of the construction (cycles 1 to 3) are not similar to cycles 10 to 12, because there is not intermediate execution on PP2 and PP3 of another iteration. Cycles 4 to 12 contain the loop that is executed $\frac{p^m}{2} = 128$ times, which is the total amount of iterations over r divided by 2 because of parallelism. As there are only $p^m - 1$ executions necessary, one too many is done. To save sequencer space, this value is calculated anyway so as to prevent it from being a special case. Cycle 1028 is added as "outtro" because it is used to write the result of the last $\mathcal{C}8$ to memory.

5.3 Decoder integration to streaming application

To make the implemented decoder feasible for deployment on actual hardware, some changes and integrations need to be made. First of all, the implementations must be adapted to accept data from outside the Montium. Secondly, both the syndrome calculator and the error locator should be fit into a single configuration. The first section will discuss communication. The second section will discuss the integration of both implementations into one.



(a) Configuration of first cycle

(b) Configuration of remaining cycles

Figure 5.10: PP configuration for component $\mathcal{A}$

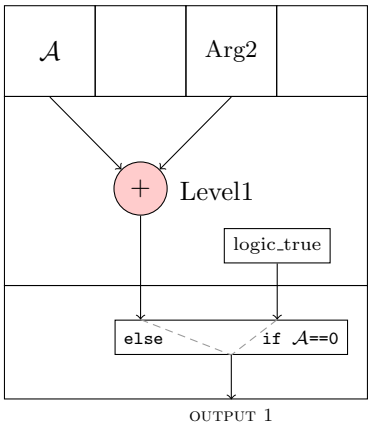


Figure 5.11: Configuration of PP for $\mathcal{B}$ and $\mathcal{B}'$ for Chien search

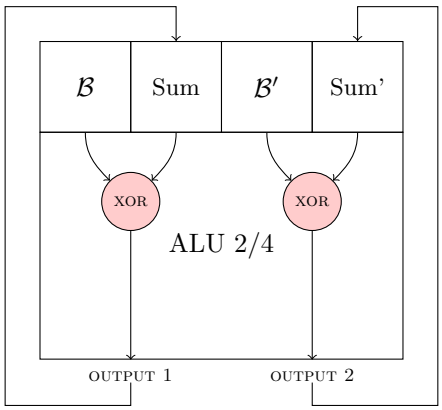


Figure 5.12: Configuration of PP for $\mathcal{C}$ of Chien search

Cycle↓	P1	P2	P3	P4	P5
1	$\mathcal{A}1$				$\mathcal{A}'1$
2	$\mathcal{A}2$	$\mathcal{B}1$		$\mathcal{B}'1$	$\mathcal{A}'2$
3	$\mathcal{A}3$	$\mathcal{B}2$		$\mathcal{B}'2$	$\mathcal{A}'3$
4	$\mathcal{A}4$	$\mathcal{B}3$	$\mathcal{C}1$ $\mathcal{C}'1$	$\mathcal{B}'3$	$\mathcal{A}'4$
5	$\mathcal{A}5$	$\mathcal{B}4$	$\mathcal{C}2$ $\mathcal{C}'2$	$\mathcal{B}'4$	$\mathcal{A}'5$
6	$\mathcal{A}6$	$\mathcal{B}5$	$\mathcal{C}3$ $\mathcal{C}'3$	$\mathcal{B}'5$	$\mathcal{A}'6$
7	$\mathcal{A}7$	$\mathcal{B}6$	$\mathcal{C}4$ $\mathcal{C}'4$	$\mathcal{B}'6$	$\mathcal{A}'7$
8	$\mathcal{A}8$	$\mathcal{B}7$	$\mathcal{C}5$ $\mathcal{C}'5$	$\mathcal{B}'7$	$\mathcal{A}'8$
9	$\mathcal{A}1$	$\mathcal{B}8$	$\mathcal{C}6$ $\mathcal{C}'6$	$\mathcal{B}'8$	$\mathcal{A}'1$
10	$\mathcal{A}2$	$\mathcal{B}1$	$\mathcal{C}7$ $\mathcal{C}'7$	$\mathcal{B}'1$	$\mathcal{A}'2$
11	$\mathcal{A}3$	$\mathcal{B}2$	$\mathcal{C}8$ $\mathcal{C}'8$	$\mathcal{B}'2$	$\mathcal{A}'3$
1028					

(128) times

Figure 5.13: Mapping of the elements of the Chien search

Streaming application

The CCU of the Montium is capable of communicating over 4 external 16-bit lanes. There are two different ways to communicate data to the Montium:

- In DMA mode. The data is sent to the Montium as address data pairs and subsequently stored into the memories. This is similar to loading a partial Montium configuration.

- In streaming mode. The data is received while the Montium is executing the sequencer code. This way the program itself has to provide for read and write instructions, and directives as to where to place the data. The program could for example store the incoming words in a memory, or use them directly as input to one of the registers of a PP.

	6.6MHz	100MHz
Syndrome calculator	1,611 kB/s	24,416 kB/s
Error locator (burst mode)	12,900 kB/s	195,320 kB/s

Table 5.5: Data transfer rate (doubled) to decoder

The disadvantage of the DMA mode is that the MONTIUM needs to be stalled in order to receive the data. When receiving large data blocks that need to be available prior to the execution of the program, this might be a desired choice. Furthermore, no input-processing sequencer instructions are required, thus saving sequencer- and configuration space. If on the other hand, the data communication takes the form of streaming, the second approach is preferred. This way it is possible to have the MONTIUM receive a value when it is actually needed by the program. The output the MONTIUM generates, can be streamed out externally as soon as it becomes available on the ALU's output.

Syndrome calculator

The input characteristic of the syndrome calculator favors the streaming approach. The design of the algorithm is specifically chosen for this approach as discussed in section 5.2. The syndrome calculator requires every symbol of every n-sized RS-block to be received in order. Every 8 clockcycles the next symbol is needed (Figure 5.9 illustrates this). Table 5.5 shows the required transfer rate of the data to the MONTIUM in kB/s. There is however a catch. The algorithm uses bytes, whereas the MONTIUM external lanes transfer data in words. If the data is to be processed immediately on arrival, this means the bytes should be sent to the MONTIUM with eight prepended zeros, to fill the whole 16 bits, effectively doubling the transfer rate of the algorithm. The values in table 5.5 are therefore doubled to prevent misconception. The resulting 16 values of the syndrome calculator reside in memories 3,4,7 and 8, all of which contain four values. As the CCU has four lanes, it will take only 4 clockcycles to output $4 \cdot 4$ values to the external lanes.

Error locator

The error locator has an input characteristic which is quite different from the syndrome calculator. The error locator only requires 1..8 of the 17 coefficients (bytes) of the Λ polynomial (the first has always value 1 and is therefore not needed). The drawback is that it repeatedly requires these bytes for its calculation. Streaming data introduces overhead and energy loss; the external system needs to repeatedly send the same data, and the MONTIUM needs to receive it. It is therefore desired to store the received bytes so reception can be limited to only once. The problem can be solved by selecting one of two possible solutions:

- Start the algorithm by receiving the 8 bytes and storing them in memory.
- Use the first iteration to receive the 8 bytes, and simultaneously storing them in memory. The next iterations shall use the values from memory.

The first approach requires a 1..8 clockcycle overhead per execution. The entire Chien search takes 1028 clockcycles, which results in an overhead of 0.10% to 0.78% clockcycles and only one internal bus configuration. The second approach requires no clockcycle overhead, but requires extra configurations. The second iteration then differs from the first, meaning extra sequencer space and bus-configurations are required. For this implementation, the first approach is selected. The main reason is that both sequencer and configuration space is saved. Although this is generally not an issue, and optimization for speed is desired, it nevertheless leaves more space for implementing and testing other parts of the RS decoder in the same configuration. Table 5.5 shows the burst speed for the both the first and second approach. In both cases the implementation requires the reception of one byte every clockcycle, for a total of 8 clockcycles. Whichever method is selected does not influence the overhead caused by a delay in reception of the data. If the bus to the Montium is not capable of delivering the bytes on time, the Montium will block and its execution will be stalled. Due to the fact that both approaches require one of the 8 bytes every clockcycle, the performance will suffer equally much at both approaches.

The output of the resulting sums is sent to the external lanes. As two separate results become available in $\mathcal{C}$ at once, two lanes can be used to handle the results. The check for a syndrome byte being zero is not performed in the Montium. To do this, the Montium should check whether the resulting root is zero, and accordingly output the accompanying value of r on the lanes. The idle cycle of PP3 is not sufficient to handle the two comparisons. Furthermore, r is only available in PP2, which then also should be available in PP3. All in all, it is not feasible to do this without sacrificing speed.

Streaming implementations

The design of the syndrome calculator and the error locator already foresee a streaming approach. They have therefore been designed by using memory 1 of PP1 from which to retrieve their data. In case of the syndrome calculator the read operation on this memory can be rewritten to a read operation from the external lane:

```
From (mov p1m1 -> destination) to (mov ext1 -> destination)
```

Writing the results to the external lane needs to happen after the results have been calculated. It can therefore be done by appending a four-clockcycle write from memories 3,4,7 and 8 to the four external lanes.

The implementation for the error locator also reads its Λ values from memory 1. Adapting the implementation to the extent in which the data is received, is straightforward. Before the execution of the implementation, 1..8 reads from the external lane are performed, and subsequently written into memory. The original implementation can then be run to calculate the error locations. Once the result of the summations become available on the output of $\mathcal{C}$ they can be sent to the external lane instead of the memory. Replacing

```
(mov p3o1 -> memory, p3o2 -> memory) by (mov p3o1 -> ext1, p3o2 -> ext2)
```

in the cycle after $\mathcal{C}8$ of figure 5.13, will make sure the summation is directed to the output of the Montium CCU.

Decoder integration

Suppose an external system (e.g. a GPP) uses the MONTIUM to calculate the syndromes and error locations of its RS packets. If the packet contains errors, and the syndrome is not found to be zero, the error locator will have to be initiated. If both implementations are kept separate, this would introduce a MONTIUM reconfiguration in between. To prevent this from happening, the entire RS MONTIUM implementation will have to reside in one configuration. As the design of both implementations take the re-usage of each others internal configurations (i.e. ALU configurations, bus configurations, etc) into account, overlap occurs which will prevent the MONTIUM from running out.

The solution for having both implementations integrated into one, is to have the external system decide which one he chooses to execute. This is achieved by letting the external system sending a zero or a one prior to the execution, which will trigger either the syndrome calculator or the error locator (Source code 5.5).

Sourcecode 5.5: Task switcher of the integrated decoder implementation

```
1 BEGIN: clock
2 mov ext1 -> p5a2
3 alu p5a2 eq 0 -> p5sb //if the bus was 1...
4 clock
5 jcc p5sb CHIEN //...jump to CHIEN search. Else continue running the
      syndrome calculator
6 clock
```

5.4 Discussion

This section will discuss the possibilities to extend the implementation to facilitate an entire Reed Solomon solution for the MONTIUM.

As can be seen in figure 5.1, there are still blocks that are not implemented. These include the calculation of the error polynomial Λ, the two error magnitude blocks, and the actual error corrector. Implementing these blocks could potentially lead to an overall improvement of performance in terms of energy consumption and speed. Furthermore, the actual external GPP will be able to perform other intermitted tasks.

Implementing remaining blocks

Every block will need to be assessed as to its ability to be implemented. Furthermore an assessment needs to be made as to how many clockcycles are required for its execution. The assessments will be made by using table 5.2 and the reference implementation. The gathered information on the implemented blocks show that this assessment can be quite accurate:

Example:

> *Table 5.2 shows that the syndrome calculator uses 4080 Galois additions and 4080 Galois multiplications. The algorithm can be executed in 16 parallel paths, giving the advantage of parallelism. The additions and multiplications only exist in MAC operations, which can be performed in one pipelined clockcycle. Furthermore it is*

proven that two MACs can be executed in parallel. Therefore the estimated number of clockcycles used for the syndrome calculator is $\frac{4080}{2} = 2040$. *The actual implementation uses 1642 clockcycles due to the described optimizations (section 5.2).*

- **Error polynomial** The error polynomial is calculated using the Berlekamp-Massey algorithm. As can be seen in table 5.2, there is no support for large sequences of parallelism. The Berlekamp-Massey algorithm introduces a fairly large amount of control, which will have a negative influence on the number of required internal Montium configurations. As for example the discrepancy calculator does support parallelism and features a MAC, the actual number of required clockcycles will be a little less than the total number of Galois additions and multiplications combined $(592 + 848 = 1440)$. Depending on the number of erasures, an amount of 1200 clockcycles seems a fair worst-case estimation.
- **Error evaluator** The error evaluator features 16 parallel paths. The additions and multiplications are executed sequentially and contain no MAC operations. There are $2048 + 1024 = 3072$ sequential calculations which can be executed in 16 parallel paths. Due to the requirement of reading and writing the polynomials to and from memory, parallelism on five ALUs is probably not possible. If only two ALUs are used, which is a safe, worst-case estimation, this will lead to $\frac{3072}{16/2} = 1536$ clockcycles being required for execution. Depending on the configuration space that is available, and the amount of memories that are required and available, a choice has to be made as to how the parallelism is actually dealt with.
- **Errata polynomial** Calculating the errata polynomial requires 392 additions and 392 multiplications, all of which are MACs. Due to the parallelism of eight, again two MAC operations can be performed at the same time on the Montium. This means that the errata polynomial can be calculated in a maximum of $\frac{392}{2} = 196$ clockcycles.
- **Error corrector** The error corrector only requires a maximum of eight additions, all of which could be done in parallel. As the input data is probably located in only one or two memories, five parallel paths can not be efficiently used. Therefore a safe estimation is made which results in a maximum number of eight clockcycles.

Table 5.6 shows the number of clockcycles of each block, which leads to an estimation of 5618 clockcycles. Memory interaction possibly introduces idle cycles or limits the way in which the algorithms can be implemented. As each estimation does not take this into account, the actual estimation should be higher. We believe it should be possible to implement a Reed Solomon decoder, if configuration space is adequate, that runs within 6500 clockcycles. Note that the precondition includes the worst case of eight errors being corrected. Also, possible reconfiguration is not taken into account.

Optimizing Chien search

The optimization of the dynamic number of λ coefficients in the Λ polynomial can increase the speed of calculation. The $v = 8$ approach requires 1028

	Clockcycles
Syndrome calculator	1642
Error polynomial	1200
Error locations	1036
Error evaluator	1536
Errata polynomial	196
Error corrector	8
Total	5618

Table 5.6: Estimated clockcycles for Reed Solomon algorithm blocks

clockcycles c, but will scale linearly:

$$c = 3 + 128v + 1 = 128v + 4 \tag{5.8}$$

This holds if the current design is implemented for a variable v. Unfortunately designing such an implementation is not trivial. Figure 5.13 looks regular, but in fact is not runtime scalable to dynamically support a certain v. Memory lookups have to be initiated up to two clockcycles before they are required, and the addressing the correct Λ coefficient requires resetting in time. This way, the smaller the v, the more the different iterations will need to overlap per clockcycle. Furthermore, runtime variable execution requires evaluating a loop counter which requires an ALU operation, or manipulation of an existing loop counter in the sequencer. This could introduce control gaps in the execution of the algorithm which can severely influence speed. Due to insufficient time and late discovery of this principle, a design for this application is not made.

6 Results

This section presents the results of the implementations on the MONTIUM. The first section discusses the implementation of the encoder, while the second section discusses the implementation of the decoder.

6.1 Encoder

The amount of required calculation clockcycles by the encoder can be found in table 6.1. As there is no external I/O implemented, these figures only indicate the actual calculation time. In order to be able to compare the results to the results in [14], also the non-standard RS(255,239) is considered. The encoder of [14] is capable of encoding an RS(255,239) block in 1920 clockcycles, which is 87% more efficient than the implementation described in this research. As our goal was to simply map an existing hardware implementation one-on-one to a MONTIUM in its most basic form, optimizations to the algorithm or calculation itself were not made. Our implementation for example introduces idle ALU cycles because of the log and exp retrieval. [14] uses these idle cycles to perform other calculations.

Our implementation is capable of calculating 2345 RS(204,188) blocks per second on a 6.6MHz MONTIUM ($\frac{86.016}{13}$MHz), and 35,448 blocks on a 100MHz MONTIUM. This leads to a calculation throughput of $3,363$ kb/s on a 6.6MHz MONTIUM, and $52,064$ kb/s on a 100MHz MONTIUM. Due to the fact that results of the encoder are of no major importance in this research, no further results are available.

	RS(204,188)	RS(255,239)
Processing-intro	1	1
Processing-loop	2820	3585
Processing-outtro	-	-
Total	2821	3586

Table 6.1: Number of clockcycles of the encoder using the MONTIUM

	Cycles	6.6MHz	100MHz
Syndrome calculator	1642	$248\mu s$	$16\mu s$
Error locator	1028	$156\mu s$	$10\mu s$

Table 6.2: Number of clockcycles, and processing time using the Montium for RS(204,188)

	Syndrome	Error loc.
Input reception		8
Processing-intro	9	3
Processing-loop	1616	1024
Processing-outtro	12	1
Output transmitting	5	0
Total	1642	1036

Table 6.3: Number of clockcycles with I/O using the Montium for RS(204,188)

6.2 Decoder

The amount of required calculation clockcycles by the syndrome calculator and the error locator can be found in table 6.2. The table also contains information about the amount of milliseconds the calculation of one RS block requires. The total amount of clockcycles consists of several sub-tasks. Table 6.3 shows the sub-tasks of both decoding processes and indicates the number of clockcycles that are used. With these figures, the performance of the Montium can be calculated.

The amount of required clockcycles can also be expressed as formulae of n and k. For the syndrome calculator calculation (excluding pre- and postcalculation I/O):

$$Cycles = \frac{n-k}{2} \cdot n + 5 \tag{6.1}$$

And for the error locator:

$$Cycles = 128t + 4 \tag{6.2}$$

Using these formulae it is possible to calculate the required cycles for any n and k, given that $n-k$ is even. This is a precondition, because of the mirrored design of the implementations.

As chapter 5.3 describes, the selection between syndrome calculator and the error locator takes two clockcycles per "choice". This is included in table 6.4, which shows the number of RS data blocks that both the syndrome calculator or the error locator can handle per second. If a continuous stream of packets contain no errors, only the syndrome decoder is invoked. If in a worst case situation, every packet contains errors, the error locator will be invoked for every packet. The Montium decoder is then capable of handling the throughput as given in table 6.5. Note that after the syndrome decoder has finished, an external process should calculate the Λ. This calculation delay is not taken into account in the table.

	6.6MHz	100MHz
Syndrome calculator	4029	60,901
Error locator	6386	97,276

Table 6.4: Number of RS blocks processed per second, with I/O, using the MONTIUM for RS(204,188)

	6.6MHz	100MHz
No errors packets	6,272 kb/s	94.79 mb/s
All errors packets	3,938 kb/s	23.94 mb/s

Table 6.5: RS throughput that the MONTIUM can handle for RS(204,188)

The decoder has been tested using the simulator. The test bench of the syndrome calculator consists of 50 packets, half of which are modified to contain errors. Special packets have been designed for specific flaws the decoder may have. This way, during an early test the $0 \cdot 0 \neq 0$ bug was found (section 4.2). As zero, and calculations that exceed the field, are in fact the only exception in the normal range, no other specific packets were designed. The tests were performed by manually writing the data into memory, running the decoder, and reading the result from memory. The final test showed no flaws or shortcomings with respect to the syndrome calculator. The error locator also has been tested. This test consists of changing the Λ coefficients, and letting the error locator search for its roots. As with the syndrome calculator, the error locator has been tested with 50 pseudo random polynomials, which were written to the memory of the MONTIUM before executing the test. Several polynomials were modified to contain zeroes at several places. The results were again read from a memory. The final test showed no flaws or shortcomings with respect to the error locator.

Comparison to other architectures

The MONTIUM has been compared to other architectures with respect to execution speed. The architectures that were used are the

- ARM946 E-S, running at 86.016MHz
- MONTIUM, running at 6.6MHz
- MONTIUM, running at 100MHz
- Personal Computer, running at 1.59GHz Intel® Pentium® M processor. It uses Microsoft Windows XP on 1GB internal memory.
- Xilinx DVB1 Reed Solomon Decoder, IP Core [7].

The figures for the ARM and the Pentium® are actually measured. To achieve this, the original C code was rewritten to contain only the actual calculation loops. It is ensured that every execution of the decoding process is unique, to make sure the compiler does not optimize by "a priori" knowledge.

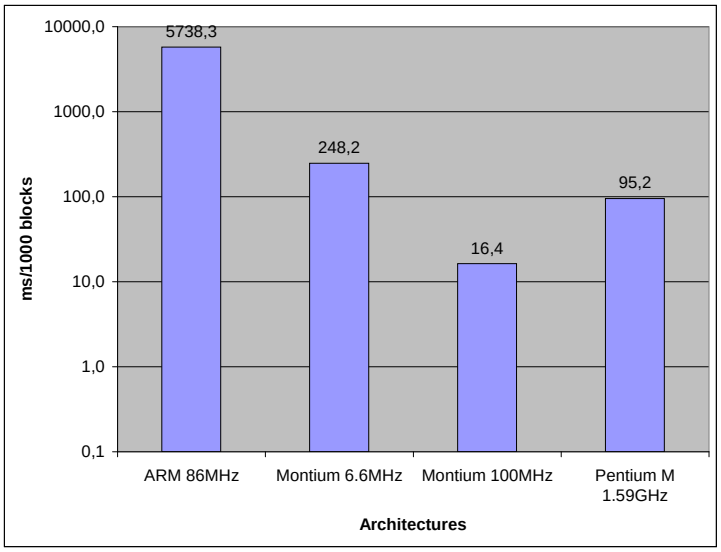


Figure 6.1: Syndrome calculator throughput speed comparison

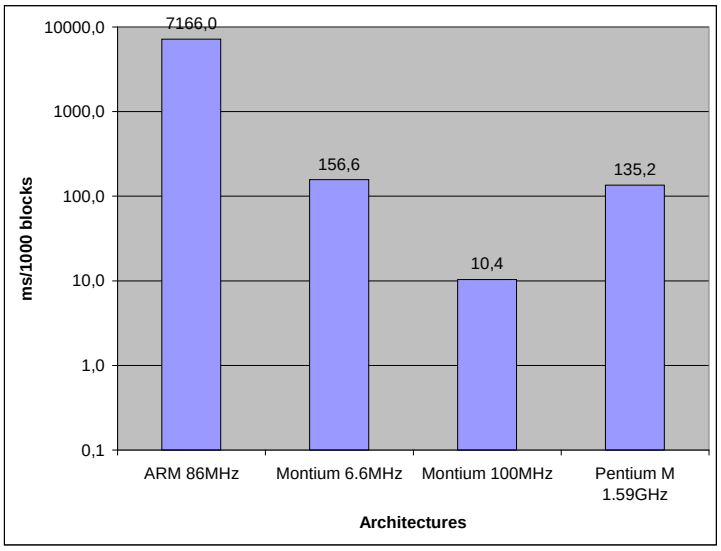


Figure 6.2: Error locator throughput speed comparison

On the PC the code was executed under Microsoft Windows XP whilst minimizing the amount of other processes. The implementation for the ARM was executed on BasOs[19]. To minimize overhead of the operating system, the scheduler has been disabled. By timing 200,000 executions of each decoding process, the calculation times were derived. The execution-time of the Montiums (including data transportation from an external architecture) are based on the assumption that there is no delay in transportation. Figure 6.1 and 6.2 show the result of the execution of 1000 invocations of the algorithms.

The Xilinx DVB1 IP Core is built by the same standard as the other decoders presented in this research [7]. This means it has the same n, k, t, has a fixed block length and processes the blocks serially. It can therefore be used in a comparison with the other architectures. The processing delay is the time between the arrival of the first symbol of a block on the input, and the arrival of the first symbol of the next block. Using the processing delay and the frequency of the hardware it runs on, enables the calculation of the throughput. The latency of the Xilinx DVB1 IP Core is 620 symbol periods [7], which results in 620 clockcycles as a symbol can be clocked in in one clock cycle. The maximum frequency is given to be 266MHz when FPGA Speed Optimizing instead of Area Optimizing is set. Comparing the Xilinx core to the individual syndrome and error locator measurements is difficult. It is only known that the syndrome decoder uses 61.5% of the calculation time on an ARM, and the error locator 26.4% (table 5.3). In 5.4 it is derived that an entire RS decoder implementation would be feasible within 6500 clockcycles on the Montium. The Xilinx core is able to decode 1000 blocks in $\frac{1}{266MHz} \cdot 620 = 2,331ms$. The 100MHz Montium will decode 1000 blocks in $\frac{1}{100MHz} \cdot 6500 = 65ms$. Figure 6.3 contains these findings.

It can be concluded that the Montium is not able to decode RS blocks as fast as an FPGA implementation. We stress that this comparison, however true, does not mean that the Montium is a bad choice. The Montium still is fast enough to decode up to 20 DMB ($\frac{23.94mb/s}{1.152mb/s}$) channels worst case, simultaneously. It is also capable of decoding according to the DVB-H standard (15 mb/s).

The Montium has been designed with specific functionality in mind, and is very energy efficient. The algorithms within the decoder that the Montium design targets, have been implemented. Together the syndrome decoder and the error locator perform 87,9% of the total amount of time of an ARM, in 3819 clockcycles. This means that the remaining 12,1% requires 2681 cycles, clearly showing that there are areas in which the Montium can not be efficiently applied. Due to suspected parallelism of calculation paths in the Xilinx core, no comparison can be made between the current implemented Montium parts and 87,9% of the calculation time of the Xilinx core.

Energy performance

This section the energy requirements are estimated, to give an idea of the differences in energy consumption. To do this, we make an estimation of the energy consumption of the ARM, the Intel® Pentium® M and the Montium.

The energy consumption of the ARM946 E-S is 0.46mW/MHz [6], or 0.46 nJ per clockcycle. As the decoder takes 5738.3ms to calculate 1000 syndromes,

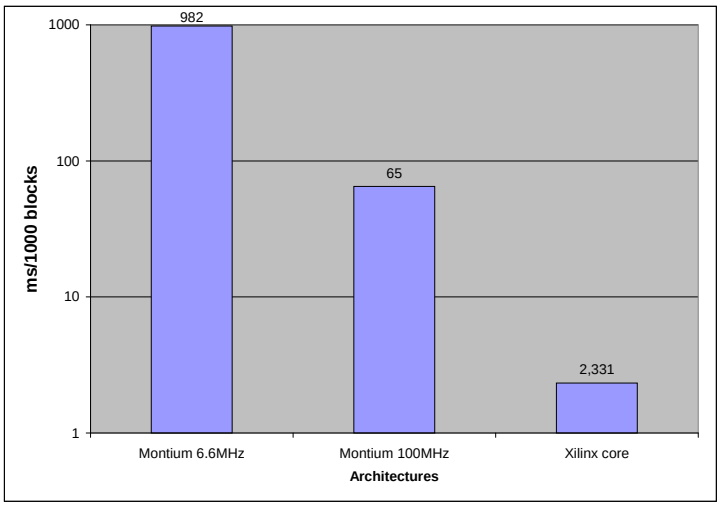


Figure 6.3: Decoder speed comparison

the energy consumption can be calculated;

$$\frac{5.7383}{1000} \cdot (86 \cdot 10^6) \cdot (0.46 \cdot 10^{-9}) = 227\mu J$$

per RS block. The energy consumption for the error locator can be calculated in the same way, and is $283\mu J$.

The energy consumption of the Montium is estimated on the basis of tests in [11]. By assuming that effective clock gating is possible, the energy consumption of the Montium is negligible when idle. The energy consumption of the Montium when calculating syndromes is estimated to be equal to the FFT64 implementation in [11]. The required resources on average resemble the memory addressing and used ALUs of the FFT64. The multiplier however is not used. Therefore, the energy consumption of the syndrome decoder is estimated to be maximally 0.500mW/MHz, or 0.500nJ per clockcycle. As the error locator is quite similar, concerning resources, the same figure is used. The energy consumption excludes the CCU of the Montium and the communication between external hardware and the Montium. The energy consumption is calculated by multiplying the energy per clockcycle by the amount of clockcycles, and can be found in table 6.6.

The energy consumption of the 1.6GHz Intel® Pentium® M processor is estimated to be 24.5W on maximum clock frequency [4]. As the PC performed $100,000$ syndrome blocks in 9514ms, its energy consumption can be calculated. The PC uses $24.5 \cdot \frac{9514}{1000} = 233.1$J for $100,000$ blocks. Therefore one block requires 2331μJ. The energy consumption for the error locator is derived in the same way, and also listed in table 6.6.

Figure 6.4 shows a graphical representation of the energy consumption.

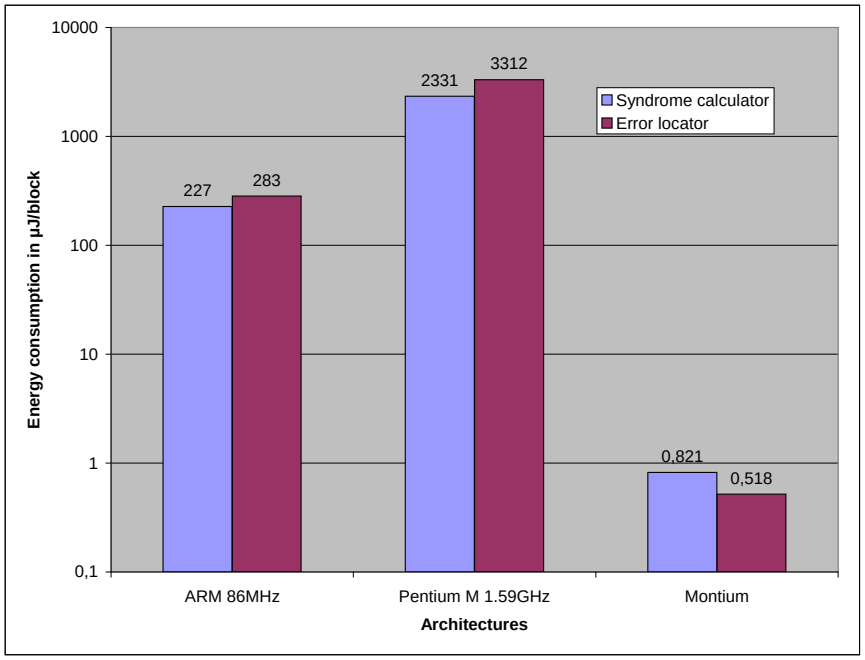


Figure 6.4: Energy consumption per RS block

	ARM	Pentium	Montium
Syndrome calculator	227μJ	2331μJ	0.821μJ
Error locator	283μJ	3312μJ	0.518μJ

Table 6.6: Energy consumption per RS block

7

Conclusions, recommendations and future work

This chapter summarizes the most important conclusions and will give some recommendations for future work.

7.1 Conclusions

1. Reed Solomon arithmetic can best be implemented in the MONTIUM by using a lookup-table. Other approaches are possible, but they perform worse considering speed and energy consumption.

2. The Reed Solomon multiplication and addition arithmetic require only one ALU configuration and clockcycle. The multiplication requires having the log values of the arguments as input, and has the log value as output.

3. It is at least possible to perform 76% of the additions and multiplications of Reed Solomon on the MONTIUM. It is also at least possible to let 87.9% of the average ARM calculation time per Reed Solomon block, be performed on the MONTIUM.

4. Simulating an LFSR on the MONTIUM is much slower than a combinatorial implemented LFSR. It is also slower than another, more optimized implementation for the MONTIUM.

5. It should be possible to have a complete Reed Solomon decoder in the MONTIUM running in maximally 6500 clockcycles per block. This assumes that eight errors are present that need to be corrected.

6. The 100MHz MONTIUM can calculate the syndromes of 60,900 blocks, or the error locations of 96,524 blocks, per second. Therefore it is able to decode up to 20 DMB channels simultaneously, and 1 channel DVB-H, worst case.

7. The MONTIUM performs the implemented algorithms much faster than the ARM or the Pentium® GPP, which both have a much higher clock frequency.

8. The Montium is much more energy efficient for the implemented algorithms, than the ARM or the Pentium® GPP, in the implemented algorithm domains.

9. The Montium is a good choice when combined with a GPP, because calculation intensive tasks can be done much faster and more energy efficient, if the communication bandwidth is sufficient.

7.2 Recommendations and future work

1. There should be looked into the possibilities of FFT based Reed Solomon decoders. The Montium performs well on FFTs and it is well possible that such algorithms could benefit overall performance.

2. There should be looked closer into the possibilities of an entire Reed Solomon implementation in the Montium. So far, only the proof of concept and initial design has been done. Also an estimation of the maximum amount of required clockcycles is performed, but it needs to be proven. The error locator should be designed so that it is dynamically capable of adapting itself to the number of errors that have occurred.

3. To gain performance in the current implementation, the implemented algorithms itself could possibly be pipelined. By pipelining for example different executions of the syndrome calculator, it is likely that clockcycles are saved.

7.3 Problem statement

> "Does the Montium TP enable an efficient implementation of Reed Solomon, with respect to speed and energy efficiency?"

The 100MHz Montium decodes correct RS packets faster than the evaluated ARM and Pentium. If errors have occurred, the calculation of the error locations is also fastest on the Montium. For the entire decoding process, assuming t errors have occurred, this is different. As the Montium requires 2681 of its 6500 cycles to calculate only 12,1% of the calculation time the ARM requires, it is easily seen that the selected ARM performs better on certain parts. The Xilinx is much faster than the Montium when it comes to clockcycles. The Montium is much more energy efficient than the ARM and Pentium, but unfortunately these figures are not available for the Xilinx implementation. The conclusion is that it is possible to create an efficient implementation of Reed Solomon, depending on the required flexibility of the application. When the algorithm should run on flexible, reconfigurable hardware, the Montium shows good energy efficiency as well as performance. However, if very high speeds are required, a dedicated solution (e.g. Xilinx) will be a better choice.

Bibliography

[1] Information technology - open systems interconnection - basic reference model: The basic model, 1994. ISO/IEC 7498-1.

[2] Ieee standard verilog hardware description language, 2001. IEEE Std 1364-2001.

[3] Ieee standard vhdl language reference manual, 2002. IEEE 1076-2002.

[4] Intel® pentium® m processor datasheet. http://download.intel.com/design/mobile/datashts/25261203.pdf, 2004. Intel.

[5] Smart chips for smart surroundings. `http://www.smart-chips.org/public/html/index.php`, 2004-2007.

[6] Arm 946 technical data. `http://www.arm.com/products/CPUs/ARM946E-S.html`, 2007. ARM.

[7] Reed solomon decoder product specification v6.1. `http://www.xilinx.com/support/documentation/ip_documentation/rs_decoder.pdf`, 2007. Xilinx.

[8] Leon Adams. Choosing the right architecture for real-time signal processing designs. Whitepaper, November 2002. Texas Instruments.

[9] Arjan C. Dam, Michel G.J. Lammertink, Kenneth C. Rovers, Johan Slagman, Arno M. Wellink, Gerard K. Rauwerda, and Gerard J.M. Smit. Hardware/software co-design applied to reed-solomon decoding for the dmb standard, 2006. University of Twente, Department of EEMCS, the Netherlands.

[10] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[11] Paul M. Heysters. *Coarse-Grained Reconfigurable Processors*. PhD thesis, University of Twente.

[12] David Joyner. Reed-solomon codes and cd encoding. `http://cadigweb.ew.usna.edu/~wdj/reed-sol.htm`, 2002.

[13] B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice-Hall, 1988.

[14] Berry Klomp. Reed-solomon error correction for wireless embedded systems, 2004. BSc Thesis, University of Twente, the Netherlands.

[15] Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, June 2005.

[16] O. Pretzel. *Error-correcting codes and finite fields*. Clarendon Press, 1992.

[17] M.Y. Rhee. *Error-correcting coding theory*. McGraw-Hill, Inc. New York, NY, USA, 1989.

[18] Thomas Ricker. Digital television: Making sense of it all. `http://www.engadget.com/2006/01/17/digital-television-part-1-making-sense-of-it-all/`, 2006. Engadget.

[19] Bas A. Van Sisseren. Design of a lightweight real-time streaming kernel. Master's thesis, University of Twente, the Netherlands, 2007.

[20] John von Neumann. First draft of a report on the edvac. Between the United States Army Ordnance Department and the University of Pennsylvania Moore School of Electrical Engineering, 1945.