Hardware/Software Co-design Applied to Reed-Solomon Decoding for the DMB Standard

Arjan C. Dam, Michel G.J. Lammertink, Kenneth C. Rovers, Johan Slagman, Arno M. Wellink,
Gerard K. Rauwerda, Gerard J.M. Smit
University of Twente, Department of EEMCS, the Netherlands

Abstract

This paper addresses the implementation of Reed-Solomon decoding for battery-powered wireless devices. The scope of this paper is constrained by the Digital Media Broadcasting (DMB). The most critical element of the Reed-Solomon algorithm is implemented on two different reconfigurable hardware architectures: an FPGA and a coarse-grained architecture: the Montium. The remaining parts are executed on an ARM processor. The results of this research show that a co-design of the ARM together with an FPGA or a Montium leads to a substantial decrease in energy consumption. The energy consumption of syndrome calculation of the Reed-Solomon decoding algorithm is estimated for an FPGA and a Montium by means of simulations. The Montium proves to be more efficient.

1. Introduction

In 1960, Irving Reed and Gus Solomon discovered a new way of mathematical error correction called Reed-Solomon coding. This new coding proved to be a very powerful algorithm to solve (burst) errors, leading to its use in countless applications ranging from digital audio discs to reliable wireless communication [14].

In the domain of wireless devices, energy consumption is a major constraint. For example, digital video broadcast decoding results in a large amount of calculations to decode the signal to correct errors. An all-software implementation of the Reed-Solomon decoding algorithm on a general-purpose processor might not be the most energy efficient solution. Therefore, possibilities and benefits of implementing parts of the algorithm onto different hardware architectures have been investigated with respect to energy consumption.

This paper describes research on energy efficiency benefits gained by implementing parts of the Reed-

Solomon algorithm in reconfigurable architectures. This has led to an algorithm execution where a general-purpose processor cooperates with reconfigurable hardware [6]. Parts of the algorithm, where parallel execution could be exploited, were (partially) implemented on a Field Programmable Gate Array [2] and a Montium Tile Processor [11]. The energy consumed on the reconfigurable hardware architectures is compared with the energy consumption on an ARM processor.

Related research on energy efficiency can be found in [22] for optimising the data path or in [21] for a single ASIC implementation of a Galois Field multiplier. Other work can be found on the analysis of Reed-Solomon decoding on a different reconfigurable architecture (MorphoSys) [19]. Completely pipelined Reed-Solomon decoding is analysed in [20].

2. Application Domain

The application domain of this research is wireless video (media) reception for a handheld device. The video signal is sent over a broadcast channel.

There are a number of possibilities for low data rate video for mobile use [7]. Digital Media Broadcasting (DMB) extends the Digital Audio Broadcasting (DAB) standard with multimedia capabilities. DMB is designed for mobile use, but inherits the low data rate limitations of a DAB channel. Digital Video Broadcasting for Handheld devices (DVB-H) extends the DVB standard to allow for a wide data rate. It adds extra error correction abilities and supports diversity antenna receivers to enable mobile use.

2.1. Reed-Solomon in DMB

Figure 1 shows the different error correction layers used in the different standards [10]. Before transmitting, a data packet is first processed by multi-protocol encapsulation (MPE) and consecutively by a Reed-Solomon (RS) encoder for burst error robustness.

Finally a convolution encoder is applied for robustness against uniformly-distributed errors. After reception, decoding is processed in reverse order. In this paper, we concentrate on Reed-Solomon decoding for DMB. A data rate of 1 Mbit/s (DMB channel) and an RS(204,188) coding are assumed (see also section 4).

Figure 1. Forward error correction (FEC) stack used in different standards

2.2. Error Rates

Wireless communication channels are subject to errors because of signal attenuation and interference. Therefore, the expected average error rate is examined to determine the required processing power for each block of the Reed-Solomon decoder.

The following aspects of the communication influence the bit error rate (BER):

- Bit rate
- Carrier-to-Noise ratio (C/N)
- Interference
- Error correction techniques

In the DVB specifications [9] a fixed BER is taken, to which the C/N and bit rate are adapted. This can also be applied to the DMB standard [8].

We are interested in the bit error rates of the input data for the Reed-Solomon decoding phase, which is the output of the convolution decoder. DVB specifications [9] state that the convolution decoder's output BER should not exceed $2 \cdot 10^{-4}$ errors per hour. After the Reed-Solomon decoding step, this should result in a quasi error free output, containing less than one uncorrected error per hour.

3. Methodology

Within this research the main question is how the energy consumption can be significantly minimized for decoding a data stream with the Reed-Solomon algorithm by a co-design of software and hardware rather than a standalone software design. The software design in this case is an implementation on an ARM processor, which is common for handheld devices. In order to perform a thorough evaluation on the hardware side, both fine-grained and coarse-grained

reconfigurable processors were examined. The co-design of software and hardware in this case is a reconfigurable chip functioning as a coprocessor serving a general-purpose processor.

3.1. Approach

A detailed examination of the application domain provides the parameters for the simulation. These parameters are based on known standards and should provide a representative estimation of energy usage in practical applications using Reed-Solomon.

As a starting point, an open source implementation of Reed-Solomon decoding [17] has been modified with the parameters used by the DMB standard [8]. The ARM simulation results (described in section 5) provide an estimation of power consumption per specific block of the Reed-Solomon decoding algorithm. This estimation is based on the most critical operations: Galois field multiplications and additions (as explained in section 4.1).

With the results acquired by the ARM simulation, an energy critical block in Reed-Solomon decoding is identified. This block (syndrome calculation) can be considered as the energy bottleneck and possibly offers options to exploit parallelism in order to reduce power consumption.

3.2. Test Setup

The complete Reed-Solomon decoding algorithm was simulated on an ARM simulator. Parts of the algorithm were simulated on a Montium simulator and a FPGA simulator. The following compilation and simulation software, and devices were used:

- Arm Developer Suite v1.2, ARM7TDMI core [4][5].
- Montium LLL compiler v01.09.05, Montium simulator, Montium v01.09 [16][18].
- Altera Quartus II Version 5.1 Build 176, Altera Stratix EP1S10B672C6 [2].

These devices were chosen because all were built using 130 nm technology and have comparable energy and speed characteristics. The ARM720T, which includes the ARM7TDMI core, cache and a MMU, has a die size of about 2.4 mm^2 [4]. The Montium Tile Processor is about the same size, 2.0 mm^2 , and is comparable to the ARM720T because it includes local memories [11]. The Stratix die size is undocumented.

4. Reed-Solomon Decoding

Reed-Solomon coding [12]-[15] is a means of forward error correction: by adding redundant information before sending data over an unreliable medium, the recipient is able to correct up to a certain number of errors.

Reed-Solomon coding operates on blocks of symbols. Such a symbol is typically represented as a byte. A block is a fixed number of symbols, to which parity symbols are appended. The number of parity symbols determines the number of errors that can be corrected in the entire block.

Figure 2. Reed-Solomon encoded block of symbols [1]

A Reed-Solomon code is usually specified as a $RS(n,k)$ code of s -bit symbols, where $n = k + 2t$. In such a code, up to t errors can be corrected. The structure of a Reed-Solomon data block is shown in Figure 2. In DMB, $RS(204,188)$ is used, so up to 8 errors can be corrected. Erasure correction is not considered in this paper.

4.1. Galois Field Arithmetic

Reed-Solomon algorithms rely on finite field or Galois field (GF) mathematics [12]. These arithmetic operations require special hardware or software functions since normal additions and multiplications cannot be used.

A Galois field can be generated using a generator polynomial; each element in the field is a power of this generator polynomial. Operations on these elements give a result that falls within the field itself. Multiple generator polynomials can generate a field with the same number of (but different) elements [12].

4.1.1. Addition. Addition in a finite field is performed by adding the polynomials and taking the coefficients modulo the prime number. In case of the prime number 2, the binary notation uses these coefficients (bits) placed after each other and addition is reduced to a bitwise XOR.

4.1.2. Multiplication. A Galois field multiplication is performed by multiplying the polynomials and taking the result modulo the generator polynomials. The modulo operation can be performed as a division by the generator with the remainder being the result.

Direct implementation of this operation is difficult; therefore different methods can be applied. The approach used is to take advantage of the fact that $\log(a \cdot b) = \log(a) + \log(b)$. For byte size symbols, a Galois field containing 256 elements is needed, known as $GF(256)$. Multiplications can be accelerated by constructing logarithm and exponent¹ tables for all 256 elements at initialization time. When two numbers are to be multiplied, they are looked up in the log table and added. The result is looked up in the exponent table, giving the result of the multiplication.

4.2. Reed-Solomon Algorithm Blocks

The Reed-Solomon decoder can be divided in several functional blocks, as shown in Figure 3 [1], [12]-[17]. For each block the number of input and output symbols are indicated.

Figure 3. Sub division of a Reed-Solomon decoder

The signal that enters the syndrome calculator is the received code word, which may contain errors. The syndrome calculator can detect errors by evaluating $2t$ equations. If all syndromes are zero, there are no errors in the code word and the other blocks are skipped, resulting in the original code word without the appended parity symbols. In case of errors, the syndromes are used to calculate the error polynomial. Once the error polynomial is available, the error locator solves the roots of this polynomial with the Chien search algorithm. The error magnitude block calculates each error's magnitude. This makes the error corrector a simple Galois field adder, adding the error magnitudes to the symbols at the locations indicated by the error locator. If the number of errors is within the limit, the original data will be the result.

Since the capacity of the channel is 1Mbps, the total length of a block is 204 ($=n$) and the symbol

¹ The exponent is the inverse of the log function.

length is 8 bit, the received number of code words per second is about 643.

An evaluation is done on the required speed, the number of GF-additions and multiplications, and the amount of possible parallelism per block. The maximum of 8 errors per block is used to analyse the worst-case scenario. At points where optimization is possible this is indicated. Table 1 lists the number of additions and multiplications per block. Additionally, the amount of parallelism is given.

Table 1. Number of Galois field additions and multiplications per Reed-Solomon block

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

4.2.1. Syndrome Calculator. Every code block must always be processed in the syndrome calculator. Two alternatives can be implemented: the Horner's scheme or the check matrix [13]. In this paper we use Horner's scheme, which has been implemented in the C-code [17] that is used to profile the ARM processor. This scheme performs 4080 ($=255 \cdot 16$) multiplications and 4080 ($=255 \cdot 16$) additions in the Galois field per code block. In hardware, this can be parallelized in 16 paths of recursive multiplications and an accumulator.

4.2.2. Error Polynomial. The modified Berlekamp-Massey algorithm calculates the error locator polynomial. All multiplications, additions and inversions are calculated sequentially, so no efficient parallel algorithm can be implemented.

4.2.3. Error Locations. The Chien search algorithm simply evaluates the error locator polynomial with all 255 possible numbers and checks whether the result is zero, which indicates that a root is found. The output is at most 8 symbols (according to the number of errors). All 255 numbers can be calculated in parallel.

4.2.4. Error Magnitude. This block needs the output of the syndrome calculator and the error polynomial. The output is an array of maximum 8 symbols, corresponding to the locations indicated by the output of the Chien search block. This block (also named the Forney algorithm [12], [13]) can be divided into two parts: the error evaluator polynomial calculation and the errata polynomial calculation.

The error evaluator polynomial calculations can be done in parallel in 16 paths. Finally, those 16 outcomes are accumulated.

Next, the errata polynomial calculation determines the actual error value from the original values with the output being the maximum of 8 symbols. For each error a parallel path can be implemented.

4.2.5. Error Corrector. The correction of the errors consists of at most 8 additions. The received code word is corrected at the error locations from the error locator and from the correction symbols from the error evaluator. Finally, the parity symbols are removed from the corrected code word.

5. Profiling on the ARM

The entire algorithm was simulated on an ARM7TDMI core, to determine the execution time of every block of the algorithm. The memory access time was not taken into account. This simulation leads to an insight which specific block could offer significant power reduction when implemented in hardware. The open-source RSCODE library [17] was adapted for profiling and to meet the DMB requirements [8].

Profiling the Reed-Solomon-code shows the following results for the several Reed-Solomon blocks:

Table 2. Relative calculation time per Reed-Solomon block on an ARM7TDMI

Block	Percentage of time
Syndrome calculation	61.5 %
Error locator	6.01%
Chien search	26.4 %
Error evaluator	5.58%
Error corrector	0.49%

The Reed-Solomon decoding of 4096 blocks takes about $2.14 \cdot 10^9$ clock cycles or 21.4 s, which is about $5.22 \cdot 10^5$ clock cycles or 5.22 ms per block. Table 2 shows that the Chien search and syndrome calculation blocks need the most time and processing power.

According to the profiling results, 61.5% of the time is spent in the syndrome calculation block. This result was obtained by random input data containing the error rate as defined by the DMB standard. Therefore, this is the most energy critical block and the first candidate for implementation in hardware. The Chien search needs more mathematical operations, but since this block is only performed in case of errors, it requires less total time and energy.

6. Hardware Architectures

The implementation of the syndrome calculation block on the ARM is compared with a implementation on a FPGA and a Montium Tile Processor (TP).

The Montium TP is a coarse-grained reconfigurable device consisting of five processing parts, a sequencer and an instruction decoding block (Figure 4). Each processing part contains an ALU, a register bank and two local memories, all operating on 16-bit words. Each ALU part has two levels, which are shown in more detail in Figure 5. The first level contains four function units capable of logical functions and basic arithmetic. The two topmost function units are connected to four register banks providing input. The lower two function units are connected to the output of the units above. The second level of the processing part contains a multiply-accumulate unit (MAC) followed by a butterfly unit (used for FFT or DCT operation). The 5 processing parts and 10 memories are connected to each other and to the outside world by 10 global busses. Each ALU level, memory or entire processing part can be turned off when not used, saving energy. [11].

Figure 4. The Montium Tile Processor Architecture

7. Results

This section contains the results of the simulations of the syndrome calculation block on the different architectures. On the FPGA and the Montium, only parts of the critical block have been implemented and simulated. Through a number of equations, the total energy usage needed for calculating the critical block on these architectures is estimated.

We assume that each addition or multiplication in parallel for a certain architecture takes approximately the same area and overhead, and uses the same amount of dynamic energy, because copies of the same implementation are used. Also we will find in section 7.1 and from [11], that the power consumption scales linearly with the clock speed (when no voltage scaling is applied). This indicates that a single operation costs a fixed amount of energy and that the required performance (operations/second) determines the power consumption.

According to the ARM profiling results, Reed-Solomon decoding of 4096 blocks takes $2.14 \cdot 10^9$ clock cycles, which means $5.22 \cdot 10^5$ clock cycles per data block. The syndrome calculation is run 61.5% of the time, taking $3.21 \cdot 10^5$ clock cycles on average. Inside syndrome calculation, 11.2% is spent on additions and 25.4% is spent on multiplications, which is 18.2% and 41.3% of the block's total time respectively, totalling 59.5%.

For the energy and power calculations the following relations are used:

$$E[J] = P \left[W = \frac{J}{s} \right] / f \left[Hz = \frac{1}{s} \right]$$

$$E_{operation} = \frac{c}{m} \cdot \frac{E_{Hz}}{p}$$

$$E_{block} = m \cdot E_{operation}$$

$$P_{block} = P_{dynamic} + P_{static} = n \cdot E_{block} + P_{static} = n \cdot m \cdot \frac{E_{Hz}}{p} + P_{static}$$

E = energy consumption (of a single operation or block)

E_{Hz} = total dynamic power consumption per Hertz (W/Hz) of a single calculation cycle

P = power (of the complete block, dynamic and static)

f = clock speed (frequency)

n = number of blocks per second to be handled

m = number of operations per block

c = number of clock cycles of m operations

$(n \cdot m)$ = number of required operations per second

p = applied parallelism

Note that we only compare dynamic energy consumption and therefore the static energy consumption is set to zero.

7.1. ARM Power Estimation

The syndrome calculation performs 4080 Galois field additions and multiplications for each block. With a power consumption of 0.200mW/MHz for typical conditions for the ARM [4] we find:

The energy for a single addition is:

$$E_{\text{addition}} = \frac{c}{m} \cdot \frac{E_{\text{Hz}}}{p} = \frac{18.2\% \cdot 3.21 \cdot 10^5}{4080} \cdot \frac{2.00 \cdot 10^{-4}}{1} = 2.87 \text{ mW / MHz}$$

Applied to the critical block of the application domain there are a number of additions per block to be processed:

$$P_{\text{addition,block}} = n \cdot m \cdot \frac{E}{p} = 643 \cdot 4080 \cdot \frac{2.87 \cdot 10^{-9}}{1} = 7.53 \text{ mW}$$

The energy for a single multiplication is:

$$E_{\text{multiplication}} = \frac{41.2\% \cdot 3.21 \cdot 10^5}{4080} \cdot \frac{0.200 \cdot 10^{-3}}{1} = 6.50 \text{ mW / MHz}$$

Applied to the critical block there are a number of multiplications per block to be handled:

$$P_{\text{multiplication,block}} = 643 \cdot 4080 \cdot \frac{6.50 \cdot 10^{-9}}{1} = 17.0 \text{ mW}$$

Since we know one critical block takes $3.21 \cdot 10^5$ clock cycles with 0.200 mW/MHz at 643 blocks per second, the complete critical block consumes:

$$P_{\text{block}} = n \cdot \frac{c}{m} \cdot \frac{E_{\text{Hz}}}{p} = 643 \cdot \frac{3.21 \cdot 10^5}{1} \cdot \frac{0.200 \cdot 10^{-3}}{1} = 41.31 \text{ mW}$$

With 643 blocks per second and $3.21 \cdot 10^5$ clock cycles per block, the ARM must run at least at about 200MHz:

$$f = \frac{n \cdot c}{p} = \frac{643 \cdot 3.21 \cdot 10^5}{1} = 206 \text{ MHz}$$

7.2. FPGA Power Estimation

In order to estimate the power consumption, an FPGA was configured with ten Galois field additions or ten Galois field multiplication blocks. Energy estimation using a toggle rate of 50% was carried out, assuming random multiplication operands. Inputs may come from either inside or outside (depending on the implementation) and therefore are not taken into account. For additions, no pipelining was implemented. For multiplication, three separate lookup tables are used. The lookup chain is pipelined [3]. The power analyzer results are stated in Table 3 and Table 4.

Table 3. FPGA power estimation results for ten adders

10 adders		
Speed (Mhz)	Dynamic power consumption (mW)	Total power consumption (mW)
25	213	400
50	425	613
75	639	826
100	852	$1.04 \cdot 10^3$

Table 4. FPGA power estimation results for ten multipliers

10 multipliers		
Speed (Mhz)	Dynamic power consumption (mW)	Total power consumption (mW)
25	303	491
50	606	794
75	910	1097
100	$1.21 \cdot 10^3$	$1.40 \cdot 10^3$

Thus, dynamic power consumption and speed scale linearly with the origin at zero. We can calculate the energy consumption for the operations by using Table 3, Table 4 and the equations from the previous section.

The energy for a single addition is:

$$E_{\text{addition}} = \frac{E_{\text{Hz}}}{p} = \frac{P}{f} = 0.85 \text{ mW / MHz} = 0.85 \text{ nJ}$$

Applied to the critical block of the application domain there are a number of additions per block to be handled:

$$P_{\text{addition,block}} = n \cdot m \cdot \frac{E}{p} = 643 \cdot 4080 \cdot \frac{8.52 \cdot 10^{-9}}{10} = 2.23 \text{ mW}$$

The energy for a single multiplication is:

$$E_{\text{multiplication}} = 1.21 \text{ mW / MHz} = 1.21 \text{ nJ}$$

Applied to the critical block there are a number of multiplications per block to be processed:

$$P_{\text{multiplication,block}} = 643 \cdot 4080 \cdot \frac{12.1 \cdot 10^{-9}}{10} = 3.18 \text{ mW}$$

The additions and multiplications consume $2.23 \text{ mW} + 3.18 \text{ mW} = 5.41 \text{ mW}$. If we take the approximation that an FPGA would perform the rest of the critical block with the same performance ratio as the ARM does (also a crude approximation), then the complete critical block would consume:

$$P_{\text{block}} = \frac{P_{\text{addition,block}} + P_{\text{multiplication,block}}}{59.5\%} = 9.10 \text{ mW}$$

7.3. Montium Power Estimation

The power usage of the Montium has been estimated by implementing a Galois field addition and multiplication.

Each ALU in the Montium has four usable inputs. The two topmost function units in level 1 of the ALU can directly perform the XOR operation. Therefore, each clock cycle two Galois field additions can be computed per ALU. The Montium has five ALUs and can thus compute 10 additions per clock cycle. Note that since only the top two functional units are needed, the memories and second level of the ALU can be disabled. This means that the critical path is very short and the Montium can run at a potentially high frequency.

The only way to estimate power consumption for the Montium is a comparison with existing power estimations, as provided in [11]. A 5-tap FIR filter uses $374.11\mu\text{W}/\text{MHz}$ while using all five ALUs. It uses only 2 of the 10 local memories consuming $28.2\mu\text{W}/\text{MHz}$. Only focussing on the addition itself, we can subtract the $28.2\mu\text{W}/\text{MHz}$ from the FIR filter energy figure. Since an XOR operation is considerably simpler than a MAC operation, the energy figure of the 5-tap FIR filter is taken as an upper-bound of $350\mu\text{W}/\text{MHz}$.

The energy per addition is:

$$E_{\text{addition}} = \frac{E_{\text{Hz}}}{p} = \frac{350 \cdot 10^{-12}}{10} = 35.0 \mu\text{W} / \text{MHz} = 35.0 \text{ pJ}$$

Applied to the critical block there are a number of additions per block to be processed:

$$P_{\text{addition,block}} = 643 \cdot 4080 \cdot \frac{350 \cdot 10^{-12}}{10} = 91.8 \mu\text{W}$$

For a Galois field multiplication on the Montium, four phases can be identified. In the first phase (“log”) each operand is provided as a memory address. In the next phase (“ALU”) the results are mapped as inputs to the ALU and added. This result is used as the address of the exponent table lookup in the third phase (“exp”). In the fourth phase the output of the exponent table is available (“output”) and at the same time two new operands can be provided for the next multiplication.

The multiplication can be mapped self-contained or pipelined. The self-contained approach uses one ALU and its two local memories. The log and exponent tables are stored in one memory. The second memory is used for the other operand. The memories can only be accessed once per clock cycle, therefore this design can only be partially pipelined. The fully pipelined approach puts the exponent table in a separate memory. The advantage is that the “log” phase and the “exp” phase can now be performed in parallel as shown in Table 5. The pipelined implementation is shown in Figure 5.

The output of a multiplication is available every clock cycle (when the pipeline is filled). Using five ALUs and ten memories, a performance of three results per clock cycle can be achieved.

The maximum performance with five ALUs and ten memories for each approach is shown in Table 6. The fully pipelined design gives the best performance. It also has the additional benefit of using only three ALUs and nine memories rather than the five ALUs and ten memories as used in the other approaches.

Table 5. Fully pipelined Galois field multiplication on the Montium architecture

Clk	1	2	3	4	5	6
1 st	log	ALU	exp	output/log	ALU	exp
2 nd		log	ALU	exp	output/log	ALU
3 rd			log	ALU	exp	output/log

Figure 5 Pipelined Galois field multiplication

Table 6 Montium performance characteristics of different multiplication implementations

Design	Performance	Normalised performance
Contained	5 results per 3 clock cycles	1.7 results/clock cycle
Pipe-lined contained	10 results per 4 clock cycles	2.5 results/clock cycle
Fully pipe-lined	3 results per clock cycle	3 results/clock cycle

The syndrome calculation performs 643 blocks per second with 4080 additions and multiplications per block for 59.5% of the time. With three results per clock cycle, the Montium has to run at least at:

$$f = \frac{n \cdot m}{p} \cdot \frac{1}{59.5\%} = \frac{643 \cdot 4080}{3} \cdot \frac{1}{59.5\%} = 1.47 \text{ MHz}$$

An FFT operation uses both levels of four ALUs and makes extensive use of all ten local memories. A Galois field multiplication is simpler and uses fewer memories. However, the FFT operation uses the memories only 2/3rd of the time. Therefore the energy consumption is estimated as the average of the 64- and

1024-point FFT, which use $541.14\mu W/MHz$ and $577.44\mu W/MHz$ respectively [11]. The average of about $550\mu W/MHz$ is used for the power estimation.

The energy per multiplication:

$$E_{multiplication} = \frac{E_{Hz}}{p} = \frac{550 \cdot 10^{-12}}{3} = 183\mu W / MHz = 183pJ$$

Applied to the critical block there are a number of multiplications per block to be handled:

$$P_{multiplication, block} = 643 \cdot 4080 \cdot \frac{550 \cdot 10^{-12}}{3} = 480\mu W$$

The additions and multiplications consume $91.8\mu W + 481\mu W = 573\mu W$. If we take on the approximation that a Montium would perform the rest of the critical block with the same performance ratio as the ARM does, then the complete critical block would consume:

$$P_{block} = \frac{P_{addition, block} + P_{multiplication, block}}{59.5\%} = \frac{573 \cdot 10^{-6}}{59.5\%} = 963\mu W$$

7.4. Comparison of the Architectures

The dynamic power consumption of the different implementations is summarized in Table 7 and depicted in Figure 6 and Figure 7.

Table 7. Energy and power consumption of basic operations on the different architectures

	ARM	FPGA	Montium
Addition (nJ)	2.870	0.852	0.0350
Multiplication (nJ)	6.495	1.213	0.183
Critical block (mW)	41.31	9.10	0.963

Figure 6. Comparison of the energy consumption per Galois field operation

Figure 7. Comparison of the dynamic power consumption of the Syndrome calculation block

It is clear that the Montium performs better than the FPGA. The ARM general purpose processor is inferior on all fronts to the hardware architectures. The differences between the architectures are in the scale of a factor 3 to 10 between the ARM and the FPGA and between the FPGA and the Montium. An exception is the low energy consumption of an addition on the Montium, which is 24 times more energy efficient than on an FPGA. For the power consumption of one syndrome calculation block, the FPGA performs 4.5 times better than the ARM and the Montium performs 9.4 times better than the FPGA and 43 times better than the ARM.

8. Research Boundaries

A number of assumptions were made to narrow down the project. These are stated in this section. Also an estimation is given about the validity of the results.

8.1. Critical Assumptions

The power usage due to communication between the different architectures has not been taken into account. It was assumed that the syndrome calculation and error correction blocks are on the same chip, allowing a single input and output for the entire Reed-Solomon decoder.

It was also assumed that the energy usage of the simulated multipliers and adders scale linearly with respect to the number of multipliers and adders used. This assumption has been used to derive an energy estimation per addition and multiplication and a power estimation per block.

For the Montium processor, no static power consumption is known. Therefore it was decided to omit the static power consumption for the ARM and FPGA as well. This will decrease the reliability of the conclusion, but it is impossible to do better due to lack of information.

Galois Field multiplications and additions have been taken as a basic operation. Profiling results for the ARM show that this is a fair estimation.

The critical block also contains control instructions apart from Galois field additions and multiplications. These control instructions take 40% of the time on the ARM. As the control instructions were not implemented on the reconfigurable devices, also 40% was used as an approximation. The Montium contains special control structures and the FPGA can implement them, making the approximation an upper bound.

8.2. Deviation Estimation

The energy figures are based on power estimation tools and upper-bound estimations instead of measurements. For the ARM processor empirical numbers from the simulation have been used. For the FPGA, a power analyzer application has been used, which takes placement of hardware and routing into account; but since this is a complex process that has a lot of different input parameters, the application tends to give poor accuracy. For the Montium earlier estimations are used. Nevertheless, we believe the results for the different architectures give a good indication which hardware/software division should be made.

9. Conclusion

Reed-Solomon decoding for the DMB standard is bounded to clear timing and energy constraints. The syndrome calculator, for example, must be very energy-efficient, since this part is always running. It is also a computationally intensive block. The part that is the most computationally intensive and, therefore, may be time-critical, is the Chien search. But, since this block is less frequent in operation, its energy consumption is less than syndrome calculation.

It is clear that the performance of the FPGA with respect to minimal power consumption significantly exceeds the ARM7TDMI core, while the Montium Tile Processor in turn significantly exceeds the FPGA. Estimations of the power consumption of the syndrome calculation block show that the FPGA is about five times more energy efficient than the ARM. The Montium is about ten times more energy-efficient than the FPGA.

In Reed-Solomon decoding, syndrome calculation seems to be the bottleneck in terms of both power consumption and computation time. Because this critical block can be computed much faster and energy-efficient on a Montium by exploiting available parallelism and locality of reference, a co-design of the Montium and the ARM can be a good solution for Reed-Solomon decoding in handheld devices.

Question remains whether more parts of the Reed-Solomon decoding chain can be computed on the Montium. The Montium needs a clock speed of approximately 1.5 MHz for the critical block. More blocks can be processed when using a higher clock speed, possibly leading to even more power savings. This is left for further research.

10. References

- [1] 4i2i Design For Communications, *Reed-Solomon Codes*, http://www.4i2i.com/reed_solomon_codes.htm
- [2] Altera, *Stratix Devices*, <http://www.altera.com/products/devices/stratix/stx-index.jsp>
- [3] Altera's Homepage, *lpm_rom Megafunction User Guide*, <http://www.altera.com>
- [4] ARM Ltd., *ARM Processor Core Overview*, <http://www.arm.com/products/CPUs/index.html>
- [5] ARM Ltd., *AXD and armsd Debuggers Guide*, RealView Developer Suite Ver. 2.1, <http://www.arm.com/products/DevTools/>
- [6] Compton, K., and Hauck, S., "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2, 2002-06, pp. 171-210
- [7] Digital Radio Information, *Comparison of the DAB, DMB & DVB-H Systems*, http://www.digitalradiotech.co.uk/dvb-h_dab_dmb.htm
- [8] European Telecommunications Standards Institute, European Broadcasting Union, *Digital Audio Broadcasting (DAB)*, ETSI TS 102 428 V1.1.1, 2005-06
- [9] European Telecommunications Standards Institute, European Broadcasting Union, *Digital Video Broadcasting (DVB)*, ETSI EN 300 744 V1.5.1, 2004-11
- [10] Digital Radio Information, *Block FEC Coding for Broadcast Applications*, http://www.digitalradiotech.co.uk/fec_coding.htm
- [11] Heysters, P.H., *Coarse-Grained Reconfigurable Processors, Flexibility meets Efficiency*, CTIT Ph.D-thesis series No. 04-66, University of Twente, NL, 2004
- [12] Lin S., and Costello D.J., *Error Control Coding Fundamentals and Applications*, Prentice-Hall, Inc, Englewood Cliffs, N.J., 1995
- [13] Poli, A., and Huguët, L., *Error Correcting Codes, Theory and Applications*, Masson and Prentice Hall International, Ltd., UK, 1992
- [14] Purser M., *Introduction to Error-Correcting Codes*, Artech House, Inc., Norwood, MA., USA, 1995
- [15] Pretzel, O., *Error-Correcting Codes and Finite Fields*, Clarendon press, Oxford, UK, 1992
- [16] Recore Systems, <http://www.recoresystems.com/>
- [17] SourceForge, *RSCode*, <http://rscode.sourceforge.net>
- [18] University of Twente, <http://chameleon.ctit.utwente.nl/>
- [19] Koochi, A., Bagherzadeh, N. and Pan, C., "A Fast Parallel Reed-Solomon Decoder On a Reconfigurable Architecture", *IEEE CODES+ISSS'03*, CA, USA, 2003-10
- [20] Lee, M.H., Choi, B.S. and Chang, J.S., "A High Speed Reed-Solomon Decoder", *IEEE Transactions on Consumer Electronics*, Vol. 41, No. 4, 1995-11
- [21] Iliev, N., Stine, J.E. and Jachimiec, N., "Parallel Programmable Finite Field GF(2^m) Multipliers", *IEEE ISVLSI'04*, Tampa, FL, USA, 2004-02
- [22] Song, L., Keshab K.P., Kuroda, I., and Nishitani T., "Hardware/Software Codesign of Finite Field Datapath for Low-Energy Reed-Solomon Codecs", *IEEE Transactions on VLSI Systems*, Vol. 8, No. 2., 2000-04, pp. 160-172