PAPER Special Section on Parallel and Distributed Computing and Networking

A Fully Programmable Reed-Solomon Decoder on a Multi-Core Processor Platform*

Bei HUANG†, Kaidi YOU†, *Nonmembers*, Yun CHEN†a), Zhiyi YU†b), *Members*, *and* Xiaoyang ZENG†, *Nonmember*

SUMMARY Reed-Solomon (RS) codes are widely used in digital communication and storage systems. Unlike usual VLSI approaches, this paper presents a high throughput fully programmable Reed-Solomon decoder on a multi-core processor. The multi-core processor platform is a 2-Dimension mesh array of Single Instruction Multiple Data (SIMD) cores, and it is well suited for digital communication applications. By fully extracting the parallelizable operations of the RS decoding process, we propose multiple optimization techniques to improve system throughput, including: task level parallelism on different cores, data level parallelism on each SIMD core, minimizing memory access, and route length minimized task mapping techniques. For RS(255, 239, 8), experimental results show that our 12-core implementation achieve a throughput of 4.35 Gbps, which is much better than several other published implementations. From the results, it is predictable that the throughput is linear with the number of cores by our approach.

key words: *parallel computing, mapping strategies, Reed Solomon decoder, multicore processor*

1. Introduction

Digital communication systems need to send and receive data accurately and reliably in the presence of noise and interference in transmitting channels. Among many possible techniques to achieve this goal, forward error correction (FEC) coding is one of the most effective and economical methods, in which the sender adds some redundant codes and the receiver can then detect and correct errors introduced by the unreliable media or networks. Reed-Solomon (RS) code is one of the most effective FEC codes, and is widely used on the storage systems (e.g. RAID 6, DVD), communication systems (e.g. cellular, broadcast, network), et al.

Most FEC decoders, including RS decoder, are normally implemented using hardwired ASICs, but they are meeting great challenges due to the rapidly proliferating of wireless radio protocols. Different protocols require different codes, and ASICs designed for one protocol cannot work for another one. Due to the inflexibility, long development cycle, and high design cost of hardwired ASICs, the idea of Software-Defined Radio (SDR) is proposed, where components that have typically been implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, digital baseband, et al.) are instead implemented using software on a personal computer or embedded computing devices [1].

Though the prospect of SDR is tempting, performance inferiority of programmable processors compared with hardwired ASICs becomes one key challenge of SDR solution for digital baseband, while the high throughput and real-time requirements of current digital baseband cannot be satisfied. To overcome this performance bottleneck, multi-core processor platform is an optimal choice, which can provide high performance through parallel computing and maintain flexibility due to its programmability [2].

The performance gained by utilization of programmable processor is highly dependent on the extraction of parallelism in algorithms. In the best case, the speedup factors can be close to the number of cores [3]. Many typical applications, however, are not easy to be fully parallelized. Thus, how to parallelize the application algorithms and how to efficiently implement them on multi-core processors, including the RS decoding algorithm, is a significant research topic.

In this paper, we propose a pure software solution of RS decoder on a multi-core processor platform. We seek for the efficient methods to optimize the program by exploring the degree of parallelism of the algorithm, rationally partitioning the algorithm into multiple tasks, and mapping different tasks into suitable cores. Imagine how attractive that digital baseband, especially decoder part, the most intensive part, can be implemented into software. That can greatly reduce the research and development price, and can be applied to many communication applications such as mobile phone, digital video broadcasting, data transmission systems, optical fiber communication.

The rest of the paper is organized as follows. In Sects. 2 and 3, fundamentals of RS codes and our multi-core processor platform are introduced, respectively. In Sect. 4 we discuss the realization of Galois field (GF) arithmetic, which is the mathematical basis of RS codes. How to develop an efficient RS decoder on a multi-core processor is discussed in Sect. 5. It is illustrated elaborately from each part of the de-

Manuscript received January 12, 2012.
Manuscript revised April 20, 2012.

†The authors are with the State Key Lab of ASIC & System, Fudan University, China.

*This work is supported by NSFC 61103008, Science and Technology Commission of Shanghai Municipality 10706200300 and Shanghai Rising-Star Program 11QA1400500, National Significant Science and Technology Projects — 03 Special 2011ZX03003-003-03, Huawei Corporation and National Science and Technology Major Project of China (No.2011ZX03003-003-03) and Shanghai Municipal Education Commission ShuGuang Plan (No.11SG07).

a) E-mail: chenyu@fudan.edu.cn
b) E-mail: zhiyiyu@fudan.edu.cn

DOI: 10.1587/transinf.E95.D.2939

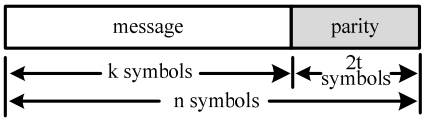


Fig. 1 The structure of one codeword block of Reed-Solomon codes.

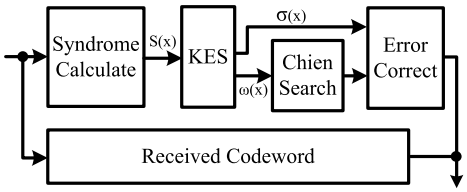


Fig. 2 Diagram of RS decoding process.

coding process, to the whole decoder on the multi-core platform. The evaluation results and the comparison are listed in Sect. 6. In the last section, conclusion is drawn.

2. Overview of Reed-Solomon Codes

The most common representation of RS codes is RS(n, k, t), which means a block of k symbols is encoded into n symbols by adding $2 \times t$ parity symbols. One encoded codeword block has the capability of correcting up to t symbol errors. Figure 1 shows the structure of one codeword block.

Practical RS codes are based on GF with 2^m elements, which will be explained in Sect. 4. In this case, each symbol can be represented as an m-bit value, and the number of symbols in one codeword block is no more than $2^m - 1$. A commonly used code RS$(255, 239, 8)$ encodes $k = 239$ message symbols into an $n = 255$ symbols block by adding 16 parity symbols, and is capable of correcting up to 8 symbol errors per block. This code is based on GF(2^8) and each symbol is represented as an 8-bit data.

The decoding process is much more complicated compared with encoding. The classical decoding process can be divided into four components, as Fig. 2 shows, including Syndrome Calculation (SC), Key Equation Solving (KES), Chien Search, and Error Correction [4].

The first step is to calculate syndrome values from one received codeword block. If all the syndrome values are zero, it can be concluded that there is no error in the codeword. Otherwise, the received codes must contain some errors.

If not all the syndrome values are zero, the second step, KES is executed to find out the Error Location Polynomial and Error Value Polynomial. The Error Location Polynomial denotes the positions of error symbols in one codeword. The Error Value Polynomial denotes error value of the corresponding symbol.

In step 3, with Error Location Polynomial from step 2, Chien Search determines the roots of the error location polynomial by checking every location to find out whether error occurs. It is an exhaustive search by substituting each of the field elements in the Error Location Polynomial. If error occurs in this location, then the forth step is executed.

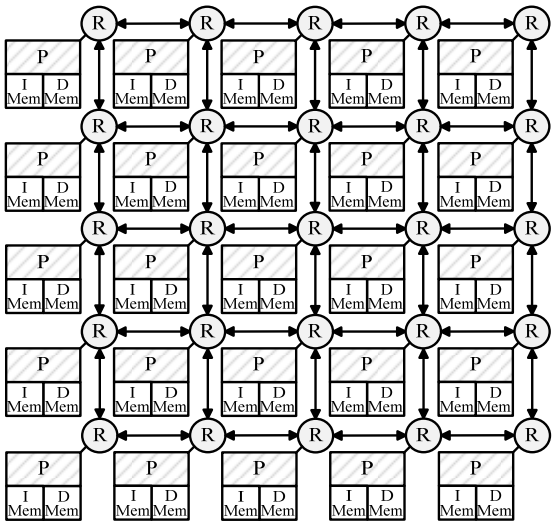


Fig. 3 Architecture of the 2-dimension mesh multi-core processor platform.

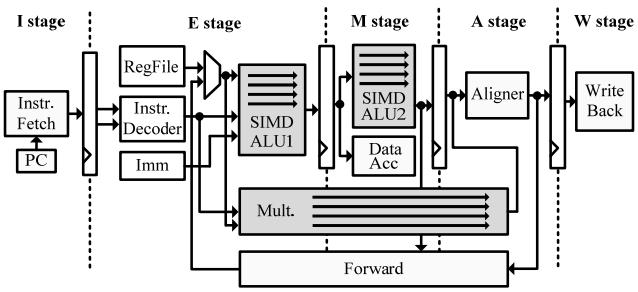


Fig. 4 5-stage pipeline of a single SIMD core.

Step 4 is calculating the error values by Forney algorithm. These error values are then added to the corresponding symbols to get the correct ones.

All these steps are GF based.

3. Architecture of Multi-Core Processor

In this section, we describe the architecture of our multi-core processor since it affects the software implementation approach significantly.

The multi-core processor employs homogeneous tiles, with 2-dimension mesh topology and message-passing communication method. It uses scalable mesochronous clocking style which allows for clock-phase-insensitive communication across tiles and synchronous operation within each tile. A 5×5 multi-core processor platform is shown in Fig. 3.

Each core in Fig. 3 is an SIMD processor enhanced from a MIPS-style processor. It has a five-stage pipeline as in Fig. 4. The I stage fetches instructions according to Program Counter (PC), and the instruction decoding is done in the first half of E stage by Decoder. With the proper data (from register file, immediate, or forward logic), SIMD ALU/Mult., which can process four 8-bit data in parallel, execute the operations in two or three cycles – two cycles

for add/simple mult. (16-bit data width multiplication) operation, three cycles for complex mult. (32-bit data width multiplication) operation. As shown in Fig. 4, second half of E stage and M stage is used for ALU; for complex mult., it needs one more cycle to finish calculation before sending result to A stage. A stage is an aligner. Forward logic is used for ALU and Mult. to alleviate data hazard pipeline penalties. In W stage, the Write Back block writes the results to register file. The MIPS-style instruction set provides high flexibility, and the added SIMD function provides higher performance for many applications especially the target communication applications. Each processor can communicate with other processors through a packet-switching network-on-chip (NoC) interface.

4. Realization of Galois Filed Arithmetic

Since the mathematical basis of RS is GF, this section presents a brief overview of GFs' properties and discusses their implementation approaches on general purpose processors.

Galois field, or finite field, is a closed set with finite elements. That is to say, the results of GF operations such as addition or multiplication are always members of the field. The algorithm of these arithmetic operations is different from the most widely recognized and used nonfinite field, the field of real numbers. GF operations are performed without carry and overflow. It means that a data can maintain m-bit even after a series of arithmetic operations have been taken. Introduction of frequently used operations [5] and their implementations are mentioned below.

4.1 Addition and Subtraction

In a finite field with characteristic 2, GF(2^m), which is widely used in communication systems, addition and subtraction are identical and can be simply accomplished by exclusive or (XOR) operation.

4.2 Multiplication

GF multiplication is a bit more complicated. It is defined as two polynomials multiplied then modulo the prime polynomial as the formula (1), here $p(x)$ denotes the prime polynomial and ".*" denotes multiplication in a finite field.

$$f(x) .* h(x) = \left(\sum_{i=0}^{m} \left(f_i \times \sum_{j=0}^{m} h_j x^{i+j} \right) \right) \% \; p(x) \tag{1}$$

GF multiplication can also be done using an exponential representation, which is much more suited for software implementations. Here each element is represented by exponential representation, α^i ($\alpha = 2$ in GF(2^m)). The multiplication is presented in (2).

$$\alpha^i .* \alpha^j = \alpha^{(i+j)\%(2^m-1)}. \tag{2}$$

Based on (1) and (2), there are four ways to do GF multiplication.

1) Pure software

This approach is to perform calculating process as (1) [6]. The advantage of this technique is that it requires little memory. The disadvantage is that it can take many processing steps to do the polynomial multiplication and the prime polynomial division, and the performance is low.

2) Full look-up table

This technique is to create a table indexed by the combination of multiplier and multiplicand whose content is the product [6]. This solution is fast compared to the pure software solution but requires large memory. For the most common GF(2^8), whose size is 256, it needs a look-up table of $256 \times 256 = 65{,}536$ bytes.

3) Partial look-up table

This approach is based on the (2). It needs two look-up tables. One table (named Log table) fills with all the 256 elements in GF(2^8) in the order of α^i ($0 \le i \le 255$). The other (named Alog table) is defined for mapping the index of Log table to its content, which satisfies alog[log[i]] = i. When processing GF multiplication, i and j are indexed by the multiplier and multiplicand from Alog table, execute some calculation (($i + j$) % ($2^m - 1$) in (2)), then get the result form Log table. This approach requires far fewer operations than the pure software approach, and far less memory amount than full look-up table approach. It is an appropriate trade-off choice.

Algorithm 1: Partial look-up table approach for GF multiplication

```
i ← Alog( α^i );   j←Alog( α^j );
k = (i+j)%(2^m-1);
result = Log(k);
```

4) Pure hardware

The pure hardware approach is to add some GF multipliers and their corresponding instructions in the core. Because GF multiplier is frequently used throughout the whole decoding process, speed of GF multiplication operation determines the performance of RS decoder. In addition, the hardware cost is low. Hence, the pure hardware approach is the most suitable option. In our implementation, we add two instruction types, GF MAC (multiplication-and-accumulation) and mult. in SIMD cores. And each type contains scalar and vector instructions. The architecture is shown in Fig. 5.

An 8-bit GF multiplier requires about 140 gates. Four multiplications (including some necessary control logic) cost about 1K gates, which is less than 1% of our SIMD core and the hardware overhead is acceptable. It takes 1 clock cycle to execute each SIMD GF instruction and has the highest performance among the four discussed solutions. Embedded GF operators can speed up the decoding process significantly with tiny overhead. Consequently, it is strongly recommended implementing GF operations with pure hardware, as EVP [7], TI DSP [8], and many other embedded processors do [9], [10].

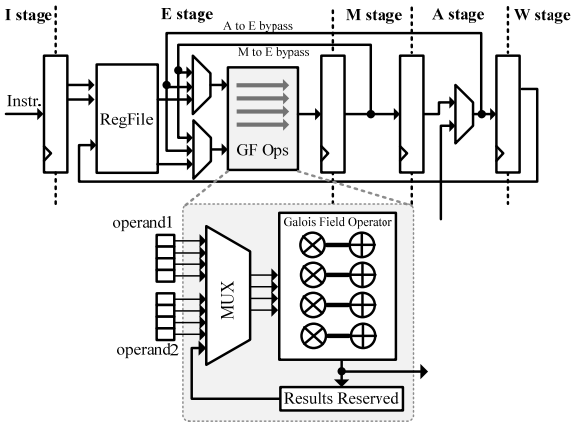


Fig. 5 The architecture of the extended SIMD core with GF operations.

4.3 Division

GF division is similar to GF multiplication. The exponential representation of GF division is in (3),

$$\alpha^i \,./\, \alpha^j = \alpha^{(i-j)\%(2^m-1)}, \tag{3}$$

where "./" denotes GF division operation.

There are four similar ways to do GF division as GF multiplication, including: pure software, full look-up table, partial look-up table, and pure hardware. Since the division is rarely used throughout the decoding process (it appears 8 times at most for RS(255, 239, 8) decoding one codeword), and has little effects to the decoding efficiency, we choose partial look-up table approach to get the best tradeoff between system performance and hardware cost.

5. Fully Programmable Implementation

This section discusses how to develop an efficient RS(255, 239, 8) decoder on our SIMD multi-core processor. Any absolute description and figures in this section are based upon RS(255, 239, 8). This RS code-rate was chosen because it is widely used as an FEC scheme in lots of radio protocols.

In each block, we first discuss how to parallelize the program on the multiple-core processor and SIMD cores to improve the performance, and then discuss how to minimize the access of memory by fully utilizing the register file since the latency and power consumption to access memory is much larger than to access register file.

5.1 Syndrome Calculation

The first step of decoding an RS code is Syndrome Calculation. An RS(n, k, t) code has $2\times t$ syndrome values, named as $S_0, S_1, S_2, \ldots, S_{2t-1}$. The following formula demonstrates how to calculate S_i $(0 \le i \le 2t-1)$,

$$S_i = (\ldots((r_{n-1}\alpha^i + r_{n-2})\alpha^i + r_{n-3})\alpha^i \ldots)\alpha^i + r_0 \tag{4}$$

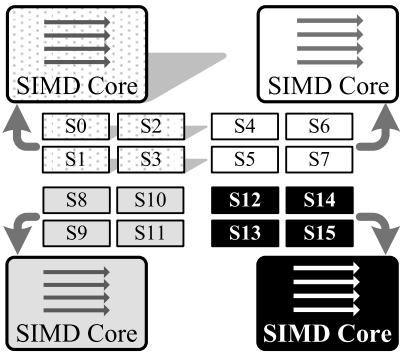


Fig. 6 The algorithm structure and mapping diagram of SC block. A group of four arrows alongside in SIMD core block means that 4 syndrome values calculation is executed in parallel.

where α^i is the element in finite field, r_j is the j^{th} symbol of the codeword [11].

Algorithm 2: Syndrome Calculation

```
for( j =0; j<=15; j++ )
   for (i = 0; i<255; i++)
      {syndrome[j]=syndrome[j] .* α^(16-j)+symbol[i]; }
```

Equation (4) is an iterative process and can be implemented using algorithm 2 above. It needs $n \times 2t = 4080$ GF MAC operations, which will be very time-consuming if sequentially calculated. Fortunately, the parallel computing capability of our multi-core platform and SIMD core can significantly improve the performance.

The key issue is how to parallelize this SC algorithm. In algorithm 2, the variables are independent among the outer loops but sharing the same codeword. It means that by dividing the outer loop the entire algorithm can be split into several parts, and the different syndrome values can be processed in parallel. Furthermore, considering the 8 bit data width of RS(255, 239, 8) and our SIMD core, four syndrome values can be calculated concurrently in one SIMD core.

As shown in Fig. 6, the 16 syndrome values are divided into four groups and mapped into four SIMD cores respectively. All these four cores work in parallel without any data exchange.

Register files are enough to hold the 2^8 bytes intermediate variables of algorithm 2. Only the codeword have to locate in the data memory which consumes 255 bytes in each SC core.

5.2 KES

The syndrome polynomial $S(x)$ is calculated in SC block, and KES block is responsible to figure out the Error Location Polynomial $\sigma(x)$ and Error Value Polynomial $\omega(x)$ by solving the equation

$$S(x) \times \sigma(x) = \omega(x) \bmod x^{2t}, \tag{5}$$

which is the key equation. Various algorithms are developed, such as ME [11], BM [12], RiBM [13] and so on.

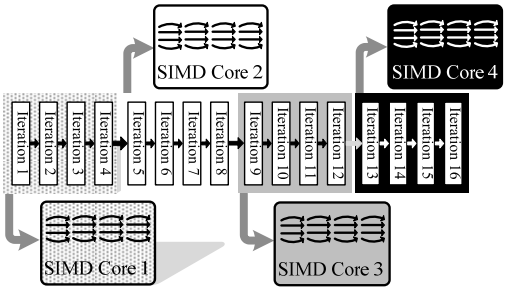


Fig. 7 The algorithm structure and mapping diagram of KES. A group of four arrows alongside in SIMD core block means that four SIMD operations are processed in parallel with regulated data stream.

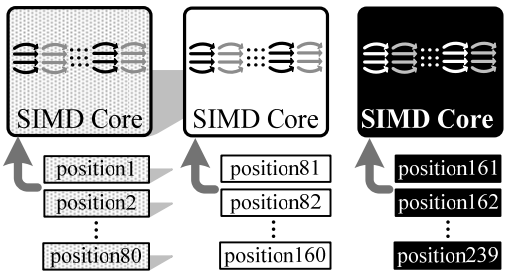


Fig. 8 The algorithm structure and mapping diagram of Chien Search block; a group of four arrows alongside in SIMD core block means that 4 positions searching is executed in parallel.

In this paper, we choose simplified degree computationless modified Euclid (S-DCME) algorithm [14], [15]. It is modified from ME algorithm to avoid calculating the polynomial degree.

For multi-core implementation, the challenge is how to partition and parallelize this algorithm because there is much dependence among the variables throughout KES process. However, multi-core processors can solve this perfectly by pipelining the tasks. Noting that S-DCME is on the whole an iterative process and the data exchange between adjacent iterations is few and regular, it is possible to take apart the 16 iterations and map them into several cores by pipelining. As Fig. 7 shows, four cores are arranged to perform KES algorithm. After completing iteration 1 to 4, core 1 sends all the needed data for core 2 to continue the iteration 5 to 8, and so on. In addition, data stream in each core is regulated to fit four parallel SIMD operations.

All the intermediate variables are stored in the register files which avoids a lot of memory load/store operations to improve the performance and reduce power consumption.

5.3 Chien Search

In Chien search, we need to search every codeword to find out the wrong ones with the Error Location Polynomial $\sigma(x)$ obtained from KES. Let $\sigma(x) = \sigma_t \times x^t + \sigma_{t-1} \times x^{t-1} + \cdots + \sigma_0$, finding the roots of $\sigma(x)$ is what Chien search does. If $\sigma(\alpha^i) = 0$, that means there is an error in this position [16]. The algorithm is described below.

Algorithm 3: Chien search

```
for i = 1 to 255
   σ(α^i) =σ_t(α^i)^t +σ_{t-1}(α^i)^{t-1}+…+σ^0;
   if (σ(α^i) ==0) → error occurs in i^th position
end
```

$\sigma(\alpha^i)$ in *algorithm 3* can be rewritten as

$$\sigma(\alpha^i) = ((\sigma_t(\alpha^i) + \sigma_{t-1})\alpha^i + \sigma_{t-2})\alpha^i + \cdots + \sigma_0, \quad (6)$$

which is a recursive process.

The basic operation of Chien search, GF MAC, is the same as SC algorithm. It needs 8 GF MAC operations to search one position.

Equation (6) is data independent between different α^is, which denote different positions. Therefore, we can map different position searching into different cores. As in Fig. 8, Chien search is mapped into three cores. Core 1 searches for position 1 to 80, core 2 searches for position 81 to 160, and core 3 searches for position 161 to 239. In addition, each core can search for four positions in parallel by making use of SIMD operations. The last 16 positions are abandoned because they are parity messages and will be discarded at last.

Chien search of RS(255, 239, 8) can be performed with no memory access by fully utilizing the 32 register files.

5.4 Error Correction

Obtaining the error location information from Chien search, Forney algorithm is employed to calculate the error value in the corresponding position. The algorithm can be illustrated in (7) [16].

$$E(k) = \frac{\alpha^k \cdot \omega(\alpha^k)}{\sigma'(\alpha^k)}, \quad (7)$$

where k denotes that an error is detected at position k, and $\omega(\alpha^k)$ and $\sigma'(\alpha^k)$ can be written into the recursive format as follows,

$$\omega(\alpha^k) = \omega_t(\alpha^{t-1})^k + \cdots + \omega_1(\alpha^1)^k + \omega_0, \quad (8)$$

and

$$\sigma'(\alpha^k) = \sigma_{t-1}(\alpha^{t-1})^k + \cdots + \sigma_3(\alpha^3)^k + \sigma_1(\alpha^1)^k. \quad (9)$$

It is worth mentioning that $\sigma'(\alpha^k)$ is obtained from Chien search, too.

Getting the error value $E(k)$, adding $E(k)$ to the corresponding symbol can finally find out the correct symbol.

According to (7), correcting one symbol needs 8 GF MAC operations and 1 GF division operation. Eight errors at most in one codeword do not cost much in our SIMD core. And the memory consumption is minimized to 255 bytes (storing one codeword) by making the best use of register files.

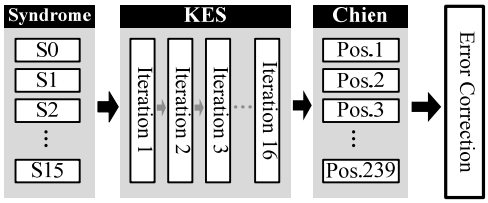


Fig. 9 The parallelizing and pipelining of the complete RS decoding algorithm.

Table 1 Number of operations and core allocation of each block.

block name	number of operations*	core allocation plan	Final core allocation
SC	4080	4	4
KES	2232	3	4
Chien Search	2629	3	3
Error Correction	80	1	1

*Operations include GF MAC, GF division, GF addition, shift and exchange. Consider the worst case.

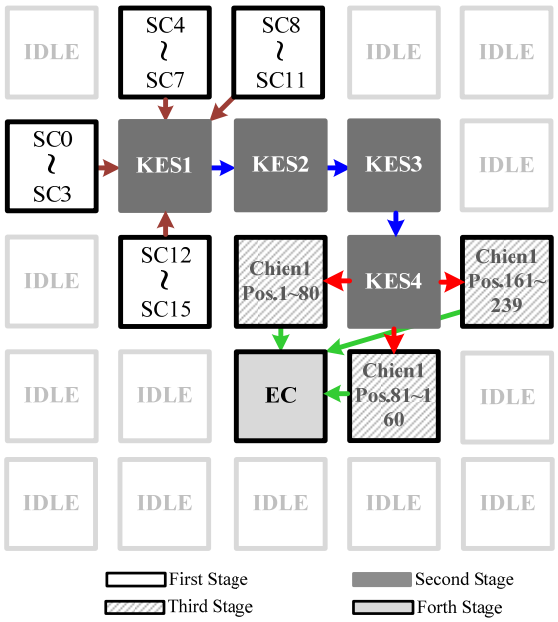


Fig. 10 Mapping diagram of RS(255, 239, 16) decoder on the 25-core processor. The gray ones with "IDLE" mark are unused and shut down.

5.5 Mapping the Whole Decoder on Multi-Core Processor

Figure 9 shows the complete RS decoder algorithm. SC block consists of 16 data independent parts. KES block consists of 16 iterations executing sequentially. Chien search block consists of 239 parts with no data exchange.

The overall mapping solution is pipelining the four blocks shown in Fig. 9 to maximize the throughput. The key issue here is how many cores should be arranged to each stage to balance the computation load in each processor core.

In the second column of Table 1, we list the number of operations of each block, and the original core allocation plan is listed in the third column with balanced computation load on each core. The final core allocation increased the cores planed for KES block, as listed in the fourth column, because this block has a lot additional exchange operations in S-DCME algorithm which are not well supported by SIMD cores where one exchange operation needs two or three instructions to complete.

Figure 10 shows the task mapping on processor cores. The mapping is based on the rules to minimize the overall latency and energy in inter-processor communication [17]. Considering the XY routing of our multi-core processor, the shorter the Manhattan distance [17] of mapping diagram, the less communication cost and latency are. The overall route length of our RS decoder mapping diagram in Fig. 10 is 16, which indicates a very good mapping scheme.

6. Evaluation and Comparison

We implement the RS decoder using C language with optimization at assembly level. The decoder runs on the RTL model of our multi-core platform.

Table 2 Execution time of all cores of processing one codeword without waiting processed data.

Processor Name	Initialization (cycles)	Send (cycles)	Receive (cycles)	Process Time (cycles)	Total Run Time (cycles)
SC1	8	2	0	318	320
SC2	8	2	0	318	320
SC3	8	2	0	318	320
SC4	8	2	0	318	320
KES1	19	19	35	248	302
KES2	0	19	19	250	286
KES3	0	19	19	250	286
KES4	0	16	19	220	265
CHIEN1	50	14	4	301	319
CHIEN2	50	14	4	301	319
CHIEN3	50	12	4	299	317
Error Correction	6	0	31	89	120

The following performance analysis is based on the 12-core mapping scheme as in Fig. 10 and in the case of pure hardware approach for GF multiplication, unless otherwise mentioned. The multi-core processor is designed to achieve 700 MHz clock frequency.

6.1 Execution Time

The execution time (counted in clock cycle) of processing one RS(255, 239, 8) codeword in the worst case (8 errors) is evaluated. The results are listed in Table 2. Here the

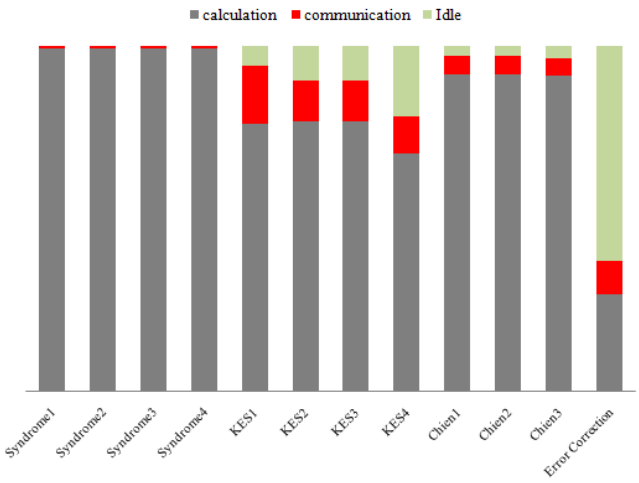


Fig. 11 The overall activity of each core implemented RS decoder.

Table 3 Throughput of the RS decoder corresponding to the occupied core number.

Occupied cores overall	number of Occupied cores				Throughput (bps)*
	SC	KES	Chien search	Error Correction	
23	8	8	6	1	8.02G
12	4	4	3	1	4.35G
7	2	2	2	1	2.13G
4	1	1	1	1	1.06G
1	--	--	--	--	420M

*All Cores are assumed to be operated at 700 MHz.

process time is the cycles spent on data processing, and the total run time includes the process time plus communication time (send and receive). SC cores have the largest execution time (320 cycles) and determine the throughput of the RS decoder.

6.2 Processor Activity Analysis

The execution time of each core shown in Table 2 doesn't include the information of the stall time waiting for inputs. The overall timing analysis of each cores' behavior including both their execution and stall time is shown in Fig. 11, which is averaged from thousands of test cases for different number of symbol errors in one codeword. The error symbols were introduced randomly.

The inter-processor communication and stall (idle) time in Fig. 11 is low, which shows the excellence of our parallelizing, pipelining and mapping strategies.

6.3 Throughput of Scalable Results

Figure 10 is not the only way for mapping an RS decoder. The decoding algorithm scales well with the number of core. When the occupied core number is increased, the throughput increases nearly linearly as some examples in Table 3. The 23-core mapping reaches throughput of 8.02 Gbps; 12-core implementation in line 2 can reach a throughput of 4.35 Gbps. Both are suitable for optical fiber communication. The single core implementation in the last line has a throughput of 420 Mbps which is prominently suitable for many applications such as mobile phone communication, data transmission technologies (e.g. DSL and WiMAX), and broadcast systems (e.g. DVB). So it is a flexible solution that we can decide using how many cores according to the throughput requirement.

Table 4 Comparison of latency and throughput under the worst case (8 errors) condition.

	Ours RS(255,239)	EVP [7] RS(255,239)	TI [8] RS(204,188)	Pal-Mult (0-p) [18] RS(255,251)	Starcore [6] RS(255,239)
Latency (cycles)	1955	787	1213	1567	14334
Data reception rate (1/throughput) (cycles)	320	787	1213	1567	14334
clock frequency (MHz)	700	300	600-800	50	300*
Throughput (Mbps)	4350	778	783-1045	65	43

*The clock frequency of Starcore is not released in [8] and is obtained from a public web.

6.4 Comparison with Others

Table 4 shows the comparison of our approach with others. The overall latency of ours, 1955 clock cycles, has little advantage than others because our core is with much fewer extended GF instructions compared with EVP [7] and TI DSP [8] and the parallelism degree of our SIMD core (4) is lower than EVP (32) and TI DSP (8). However, the throughput of our implementation can reach 4350 Mbps in the worst case of 8-error incoming codeword under a 700 MHz operation clock frequency, which is much better than others. The high throughput is the result of our multiple optimization techniques including well partitioning, pipelining, and mapping strategies. Each core in our processor is much simpler than others so that the overall system cost still remains in a low level with multiple cores.

7. Conclusion

In this paper, we present a fully software implementation of an RS decoder on a multi-core processor platform. Our multi-core processor provides significant advantages over traditional CPUs and DSPs. A high throughput RS decoder is realized on the multi-core processor by fully utilizing its parallelism and minimizing the inter-processor communication cost. The parallelism techniques include task-level parallelism, pipelining on many cores and data-level paral-

lelism on SIMD cores. The decoder can achieve a throughput of 4.35 Gbps at the worst case of the incoming codeword with each core runs at 700 MHz. In summary, the proposed software implementation of RS decoder will have wide applications in many communication protocols which require RS code.

Acknowledgment

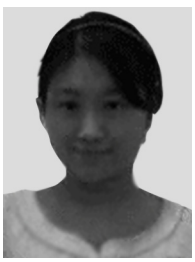
The authors gratefully thank our co-workers, Shuangqu Huang, Ruijin Xiao, Heng Quan, Zewen Shi, Yan Ying, Lin Dai, Xingxing Zhang, for a productive collaboration and helpful discussion.

References

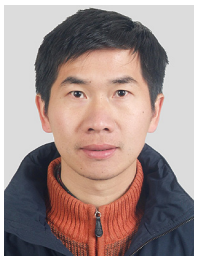
[1] W.H.W. Tuttlebee, Software Defined Radio: Origins, drivers, and international perspectives, 3rd ed., pp.3–17, John Wiley & Sons, Hoboken, 2002.
[2] Z. Yu, M.J. Meeuwsen, O. Sattari, M. Lai, J.W. Webb, E.W. Work, D. Truong, T. Mohsenin, and B.M. Baas, "AsAP: An asynchronous array of simple processors for DSP applications," IEEE Journal of Solid-State Circuits, vol.43, no.3, pp.695–705, 2008.
[3] D.P. Rodgers, "Improvements in multiprocessor system design," ACM SIGARCH Computer Architecture News archive, vol.13, no.3, pp.225–231, 1985.
[4] S.B. Wicker, V.K. Bhargava, Reed-Solomon Codes and Their Applications, pp.12–240, John Wiley & Sons, Hoboken, 1999.
[5] S. Lin, Error Control Coding: Fundamentals and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1983.
[6] D. Taipale, I.E. Scheiwe, and T.M. Redheendran, "Reed Solomon decoding on the StarCore processor," Motorola Semiconductors, AN1841/D, May 2000.
[7] A. Kumar1 and K. van Berkel, "Vectorization of Reed Solomon decoding and mapping on the EVP," Design, Automation and Test in Europe, pp.450–455, 2008.
[8] J. Sankaran, "Reed Solomon decoder: TMS320C64x implementation," Texas Instruments, SPRA686, Dec. 2000.
[9] H.M. Ji, "An optimized processor for fast Reed-Solomon encoding and decoding," IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '02), vol.3, pp.3097–3100, 2002.
[10] A. Koohi, N. Bagherzadeh, and C. Pan, "A fast parallel Reed-Solomon decoder on a reconfigurable architecture," First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp.59–64, 2003.
[11] H.M. Shao, T.K. Truong, L.J. Deutsch, J.H. Yuen, and I.S. Reed, "A VLSI design of a pipeline Reed-Solomon decoder," IEEE Trans. Comput., pp.393–397, 1985.
[12] E.R. Berlekamp, "Bit-serial Reed-Solomon encoders," IEEE Trans. Inf. Theory, vol.IT-28, no.6, pp.869–874, Nov. 1982.
[13] D.V. Sarwate and N.R. Shanbhag, "High-speed architectures for Reed-Solomon decoders," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.9, pp.641–655, Oct. 2001.
[14] J. Baek and M.H. Sunwoo, "Low hardware complexity key equation solver chip for Reed-Solomon decoders," IEEE Asian Solid-State Circuits Conference, pp.51–54, 2007.
[15] J. Baek and M.H. Sunwoo, "Simplified degree computationless modified Euclid's algorithm and its architecture," IEEE International Symposium on Circuits and Systems, pp.905–908, 2007.
[16] H. Lee, "High-speed VLSI architecture for parallel Reed-Solomon decoder," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.11, no,2, pp.288–294, April 2003.
[17] J. Hu and R. Marculescu, "Energy-and performance-aware mapping for regular NoC architectures," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.24, no.4, pp.551–562, 2005.
[18] L. Song, K.K. Parhi, I. Kuroda, and T. Nishitani, "Hardware/software codesign of finite field datapath for low-energy Reed-Solomon codecs," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.8, no. 2, pp.160–172, 2000.

Bei Huang received the B.S. degree in Electronic Science & Technology from Tianjin University, in 2008 and received the M.S. degree in Microelectronics at the ASIC & System State Key Lab in Fudan University in 2011, where she is a member of the Multi-core Processor Research & Develop group. Her research interests include the communication system and their VLSI architecture design, in particular, the channel coding and decoding implementations in both VLSI and multi-core platform.

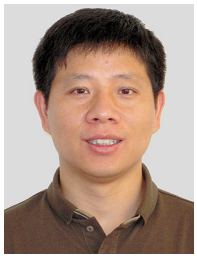
Kaidi You received the B.S. degree in microelectronics from Fudan University China in 2008, and received his master degree in microelectronics at the ASIC & System State Key Lab of Fudan University in 2011. His research interests include RISC processor and multi-processor SoC.



Yun Chen received the B.S. and M.S. degrees in microelectronics from UESTC, China, in 2001 and 2004, respectively, and the Ph.D. degree from Fudan University in 2007. Dr. Chen is currently a lecturer with the State Key Laboratory of ASIC & System, Microelectronics Department, Fudan University, Shanghai, China. Her research interests include VLSI architectures and integrated circuit (IC) design for communications and digital signal processing systems. She serves as a member of the Technical Program Committee of the IEEE International Conference on ASIC in 2009.



Zhiyi Yu received the B.S. and M.S. degrees in electrical engineering from Fudan University, Shanghai, China, in 2000 and 2003, respectively, and the Ph.D. degree in electrical and computer engineering from the University of California, Davis in 2007. From 2007 to 2008, he was with IntellaSys Corporation, CA, USA, where he participated in the design of the many-core SEAForth chips which utilize stack-based processors with extremely small area and low power consumption. In January 2009 he joined the State Key Laboratory of ASIC & System, Microelectronics Department, Fudan University, China, where he is now an associate professor. His research interests include high-performance and energy-efficient digital VLSI design with an emphasis on many-core processors. He serves as a member of the Technical Program Committee of the IEEE Asian Solid-State Circuits Conference (ASSCC) and a member of the TPC of the IEEE International Conference on ASIC. He has published 1 book and over 20 papers, and has applied 3 patents.



Xiaoyang Zeng received the B.S. degree from Xiangtan University, Xiangtan, China, in 1992, and the Ph.D. degree from Changchun Institute of Optics, Fine Mechanics, and Physics, Chinese Academy of Sciences, Changchun, China, in 2001. From 2001 to 2003, he was a Postdoctoral Researcher with Fudan University, Shanghai, China. Then, he joined the State Key Laboratory of ASIC and System, Fudan University, as an Associate Professor, where he is currently a Full Professor and the Director. His research interests include information security chip design, system-on-chip platforms, and VLSI implementation of digital signal processing and communication systems.