AN ARCHITECTURAL STUDY OF A DIGITAL SIGNAL PROCESSOR FOR BLOCK CODES

Wolfram Drescher, Menno Mennenga, and Gerhard Fettweis

Mobile Communications Systems
Dresden University of Technology, D-01062 Dresden
email: drescher, mennenga, fettweis@ifn.et.tu-dresden.de

ABSTRACT

This paper examines architectural issues for a domain specific digital signal processor (DS-DSP) which is capable of fast decoding of block codes. In real time systems it was not possible before to employ common processors for this task because of a lack of architectural and arithmetical support. We proposed solutions for the arithmetical problem in previous work [6]. In this paper we focus on architectures for implementation of different block decoding algorithms on a new DS-DSP architecture. The paper also contains benchmarks for our architecture for some selected codes and compares our DS-DSP to common digital signal processors (DSP) and dedicated logic solutions. [1]

1. INTRODUCTION

Today digital signal processors (DSP) present a key technology. The performance of DSPs is evolving further by development of new architectures rather than by advances in semiconductor technology. Therefore we measure a gain in performance by a reduced MIPS requirement per algorithm. In [1] DSPs are divided into three classes:

(1) general purpose DSP (GP-DSP)
(2) domain specific DSP (DS-DSP)
(3) application specific DSP (AS-DSP).

AS-DSPs are typically customized to a specific high-end application and DS-DSPs are defined as targeted for a wider application domain. Examples for the AS-DSP class can be found in [2] and for the DS-DSP class in [3] and [4]. The work described in this paper reflects our research from the CATS project [1], [5] of an integrated DS-DSP design system. Target of the processor described next is the domain of block codes, particularly binary and non-binary BCH codes. That requires a dedicated datapath, bus, and memory architecture.

Present software implementations of applications from mobile communications or data storage technology do not include block codes because of the required dedicated arithmetical units for real-time systems. Moreover, committees setting standards for future applications know that such codes cannot be mapped onto binary processing architectures. Probably this is one reason why those algorithms are not used for applications that demand high volume production such as cellular phones. If both finite field arithmetic and binary arithmetic were utilized for DSP technology and DSP architecture supported the algorithmic requirements of block codes we anticipate a breakthrough even for low budget applications. In chapter 2. we start from the arithmetic problem (data manipulation problem) of our target coding algorithms and develop a datapath architecture. In chapter 3. we investigate dataflow requirements (data transport problem) and propose architectures of parallel datapaths for very high throughput applications.

1.This work was sponsored in part by the Deutsche Forschungsgemeinschaft (DFG) within the Sonderforschungsbereich SFB 358

2. ALGORITHM AND ARCHITECTURE SELECTION

In [6] we proposed a MAC unit for both binary and Galois arithmetic. In this section, we discuss serial execution of a BCH decoding algorithm and derive hardware modifications of such a MAC unit for simple and fast computation of typical decoding operations.

2.1 Syndromes

Assume $\boldsymbol{r}(X) = r_0 + r_1X + r_2X^2 + \ldots + r_{n-1}X^{n-1}$ is the received code word polynomial in a *t*-error-correcting BCH code of length $n = 2^m - 1$. Let α be the primitive element of $r_i \in GF(2^m)$. The *i*th component of the syndrome of ***r(X)*** is

$$S_i = \boldsymbol{r}(\alpha^i) = \sum_{j=0}^{n-1} r_i \alpha^{ji} \qquad (1)$$

where $1 \le i \le 2t$ and $r_i \in GF(2^m)$. Eq.(1) can be written in the form of Horner's method:

$$S_i = r_0 + \alpha^i(r_1 + \alpha^i(r_2 + \alpha^i(\ldots))) \qquad (2)$$

Eq.(2) shows that the calculation of the syndromes can be reduced to multiply-add operations. However, they do not constitute MAC operations. Straightforward implementation of eq. (2) results in a structure shown in Fig. 1. Operand 1 (Op1) corresponds with α^i and Operand 2 (Op2) corresponds with r_i. We call this unit accumulate-multiply unit (ACM) because the contents of the register is used as the input for the multiplier.

The structure of the ACM unit can be merged, on the logic level, with a MAC by adding additional wiring and control gates that will flexibly direct the data using an extended bus structure. This is possible even without considerably extending die area. The reason for this is that the multiplier which takes the most area is the same for MAC and ACM. The extended MAC unit is shown in Fig. 1 (b).

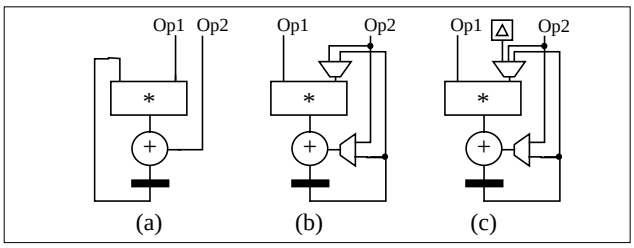

Fig. 1 ACM unit (a), and extended MACs (b), (c)

Note that this structure has the capability to perform binary MAC operations just as today's fixed-point DSPs. In addition, it can be used to calculate the syndrome of a code word using Galois arithmetic.

2.2 Key Equation

The next step in decoding a BCH code word is to find the linear feedback shift register of minimum length t that will output the syndromes if it is initialized with the first t syndromes. Let the register be described by the polynomial $\sigma(X) = \sigma_0 + \sigma_1 X + \ldots + \sigma_t X^t$ with $\sigma_i \in GF(2^m)$. An effective algorithm to find $\sigma(X)$ is the Berlekamp-Massey Algorithm (BMA) [9]. We will shortly discuss the arithmetical operations that are performed during the execution of this iterative algorithm. Then we derive the structures of the functional units that implement these operations.

At each iteration, the coefficients σ_i may be updated and the degree t of $\sigma(X)$ may be incremented. Let $\sigma^{(\mu)}(X)$ and $\sigma^{(\rho)}(X)$ be the polynomial at iteration μ and ρ, respectively, where $d_\rho \neq 0$ and $\rho < \mu$ is such that $\rho - t^{(\rho)}$ is maximum over all iterations. The update procedure is carried out as follows. First, calculate

$$d_\mu = S_{\mu+1} + \sigma_1^{(\mu)} S_\mu + \sigma_2^{(\mu)} S_{\mu-1} + \ldots + \sigma_{t^{(\mu)}}^{(\mu)} S_{\mu+1-t^{(\mu)}} \qquad (3)$$

If $d_\mu \neq 0$, calculate the scaling factor

$$\Delta = d_\mu / d_\rho \qquad (4)$$

and the coefficients of the updated error locator polynomial

$$\sigma_i^{(\mu+1)} = \sigma_i^{(\mu)} + \sigma_{i+(\mu-\rho)}^{(\rho)} \Delta \qquad (5)$$

where $i = 1, 2, \ldots, t^{(\mu+1)}$, and $t^{(\mu+1)} = max(t^{(\mu)}, t^{(\rho)} + \mu - \rho)$

If $d_\mu = 0$, take $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X)$ and $t^{(\mu+1)} = t^{(\mu)}$.

Eq.(3) is a MAC operation, hence this part of the BMA can be straightforwardly implemented on the functional unit shown in Fig. 1 (b). The execution of eq. (4) requires the division of two Galois elements. Division of two Galois field elements can be reduced to multiplying the dividend with the inverse of the divisor. The inverse be found using look-up tables. Eq. (5) constitutes a simple multiply-add operation as in a MAC unit except that the result of this one-cycle operation is not accumulated. Thus, this operation can be performed on a MAC when three operands are fetched during each clock cycle. The disadvantage of this implementation is that it would require more memory bandwidth than the other operations. However, this problem can be circumvented with the introduction of a second register (in addition to the accumulator) holding the constant Δ. Then, after the constant register has been initialized, only two operands need to be fetched thus reducing memory bandwidth to two operands per clock cycle. The functional unit that implements the necessary operations both for the syndrome calculation and the BMA is shown in Fig. 1 (c).

2.3 Error Locations and Error Values

The last step of the decoding algorithm is to find the error location numbers. This involves finding the location of the errors, and finding the values of the errors at these locations.

A procedure to detect the erroneous locations in a code word was found by Chien [10]. If

$$\sigma(\alpha^l) = 0 \qquad (6)$$

then r_{n-l} was received in error. Thus, the Chien search is performed by substituting $1, \alpha, \alpha^2, \ldots, \alpha^n$ into $\sigma(X)$ and checking if the result is zero. By applying the same approach as in section 2.1, we find

$$\sigma(\alpha^l) = \sigma_0 + \alpha^l(\sigma_1 + \alpha^l(\sigma_2 + \alpha^l(\ldots))) \qquad (7)$$

Eq. (7) shows that the Chien search requires ACM operations. Thus it can be performed with our proposed functional unit, since it already compromises ACM functionality.

After the error locations have been identified, for non-binary BCH codes, such as Reed-Solomon codes, the error values need to be found. This requires both MAC operations and divisions as well as application of Horner's method [9]. Since our extended MAC unit supports all these operations we do not discuss this further.

3. PARALLELISATION EMPLOYING MULTIPLE DATAPATH UNITS

To process the enormous amount of data in future real-time applications on one processor we need to distribute arithmetic operations to parallel units on the DS-DSP datapath. Many opportunities for parallelization at the algorithmic level exist. We can process a number of code words in parallel in different execution units or we can distribute parts of the entire algorithm to different units as it is possible for the syndromes of one code word. We do not execute loop operations of one closed set of data employing different execution units to keep the processors bus architecture simple.

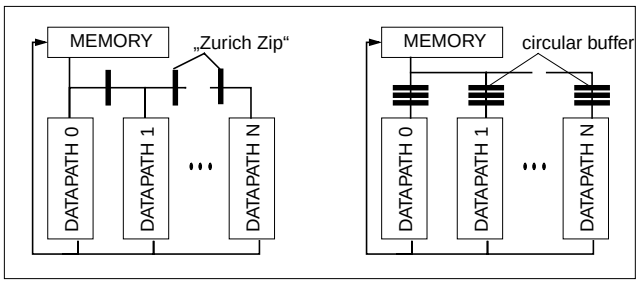

Fig. 2 Data distribution parallelization

The algorithm parallelization requires a parallel data distribution over multiple datapaths under the constraint of a low memory bandwidth. One method to achieve this is to introduce a „Zurich Zip“ register to delay input data of one unit for one cycle after distributing same data to another datapath unit. In [5] an architecture for a GSM full-rate speech codec with multiple datapaths and an

extremely low memory bandwidth based on multiple „Zurich-Zip“ registers was introduced. In highly parallel systems this approach needs some cycles to fill and to clean the „Zurich Zip“ pipeline. However, for short loops as we have in key equation and error location algorithms this is a disadvantage. For this reason we prefer to distribute parallel data by a dedicated register file or circular buffer. Each register can be loaded in advance with coefficients that are repeatedly needed in the loops. Fig. 2 depicts the two parallelization methods

3.1 Syndromes

Parallelization of syndrome calculation is a key issue because it consumes the most cycles among the three decoding steps. Working on one code word we can calculate all syndromes on $2t$ datapaths in parallel. This has the disadvantage to require an n-symbols deep circular buffer for each datapath containing all α^{ji} and all dedicated r_i from eq.(1).

A much smaller memory bandwidth can be achieved by calculating all α^{ji} in one unit (MAC1) and accumulating $\sum_i r_i \alpha^{ji}$ in a second corresponding unit (MAC0) as shown in Fig. 3. Accumulator is initialized with α^{j}.

For binary BCH codes MAC 0 can be reduced to an AND/XOR logical block and can even be appended to MAC 1. Then only one MAC-unit is needed for each syndrome having a memory bandwidth of one bit/cycle.

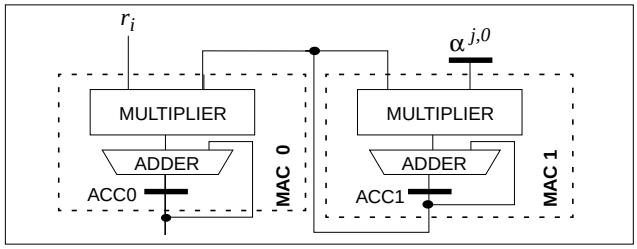

Fig. 3 Syndrome parallelization for non-binary BCH codes

Another opportunity is to work on syndrome k of c consecutive code words in parallel.

$$\begin{aligned} S_{1,k} &= r_{1,k,0}(\alpha^1)^1 + r_{1,k,1}(\alpha^2)^1 + \ldots + r_{1,k,n-1}(\alpha^{n-1})^1 \\ S_{2,k} &= r_{2,k,0}(\alpha^1)^1 + r_{2,k,1}(\alpha^2)^1 + \ldots + r_{2,k,n-1}(\alpha^{n-1})^1 \\ &\bullet\bullet\bullet \\ S_{c,k} &= r_{c,k,0}(\alpha^1)^1 + r_{c,k,1}(\alpha^2)^1 + \ldots + r_{c,k,n-1}(\alpha^{n-1})^1 \end{aligned} \qquad (8)$$

This approach leads to a low memory bandwidth. We only need one circular buffer for the parallel processed α^{jk}. In case binary BCH codes are decoded we work on symbols $r_{i,k} \in \{0,1\}$ from GF2 and α^{jk} from GF(2^m). Then (1) can be written in a logical form:

$$S_i = r(\alpha^i) = XOR_{j=0}^{n-1} (r_i \, AND \, \alpha^{ji}) \qquad (9)$$

Based on (8),(9) we designed a special 16-bit MAC unit where we integrated a syndrome accelerator employing same cells and same cell interconnections as the binary/Galois MAC uses. Two code symbols $r_{c,i}$ of c code words can be processed in parallel. The partial product compression scheme of the multiplier unit was partitioned such that it would fit to two 8-bit Galois multipliers and to the syndrome accelerators AND-part. The shifter that is used for number format switching in a binary MAC was extended by the distribution scheme for the intermediate results $r_i \cdot \alpha^{ji}$. The one‘s complement part in binary number mode used to subtract from the accumulator performs the addition of $r_i \cdot \alpha^{ji} + r_{i+1} \cdot \alpha^{j(i+1)}$. The final adder stage works as a Galois-adder. For binary numbers the accumulator-register can hold 32-bit values. We use this accumulator to store four 8-bit syndromes.

The hardware overhead to a common MAC unit is a few multiplexer which serve to distribute the data through the shifter unit to the dedicated part of the accumulator. Fig. 4 depicts a block diagram of the proposed unit and marks the corresponding structural parts of the binary MAC unit

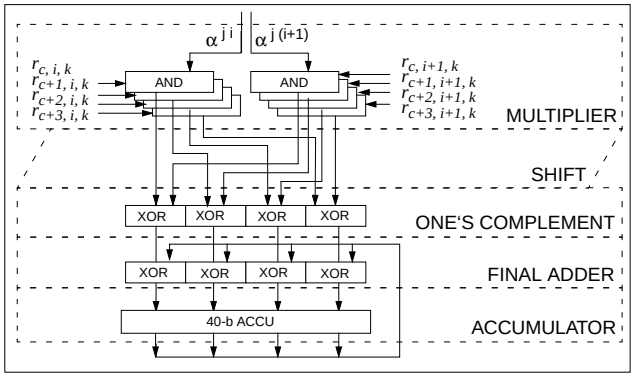

Fig. 4 Binary BCH syndrome accelerator

3.2 Key Equation

Historically, for dedicated logic implementations the BMA-algorithm was used because its recursive nature can be implemented in a strait forward manner. Unfortunately neither the BMA cannot be applied to parallel code words nor to parallel processed algorithm parts. In other words, to solve the key equation by BMA we only can involve one MAC even though we had more than one available. Employing the Euclidean algorithm [8] we achieve better results. This recursive algorithm of infinite length mainly performs polynomial division of two polynomials with coefficients from a Galois field. The initial polynomial is the „syndrome polynomial“ where each syndrome is one of the $2t$ coefficients. In a multiple MAC architecture as one coefficient of the „syndrome polynomial“can be hold in each MAC, i.e the Euclidean algorithm is best applied having $2t$ m-bit-MAC units (2^m-1 is the size of the target Galois field). If a separate Galois field inversion unit as proposed in [7] is implemented in addition an even better cycle count can be achieved.

3.3 Error Locations and Error Values

Error locations can be found in parallel by assigning N MACs to N code word symbols and performing the Chien search loop over max. t coefficients of $\sigma(x)$. A register file holds all t α-powers. Error values for binary BCH codes are obtained by checking the zero flag after the Chien search procedure. This can be supported by a hardware mechanism that checks all parallel MAC zero flags and performs a XOR operation with the corresponding part of the code word at the position where the z-flag is not set. However, for non-binary codes the algorithms are more complicated but they

can be parallelized as well by assigning one code word symbol to one MAC.

4. BENCHMARKS

Even though benchmarks are argumentative instruments because they may be corrupt we use them to compare our architecture to existing target architectures. In Fig. 5 we presume equal assumptions, i.e. the cycle count was calculated employing the best fitting algorithmic and arithmetic approach for each target hardware architecture. We assumed:

- the maximum number of correctable errors in the received code word
- received code vector consists of „high“-bits only (important for binary BCH syndrome calculation)
- finite field multiplication on binary architectures is led back to AND-XOR-SHIFT operations
- finite field inversion was done by table lookup
- a-powers were precalculated and stored in memory
- 16-MAC architecture had special hardware support for binary BCH code syndrome calculation (4 input bit executed in 1 MAC/cycle)
- up to 4 code words can be processed in parallel.

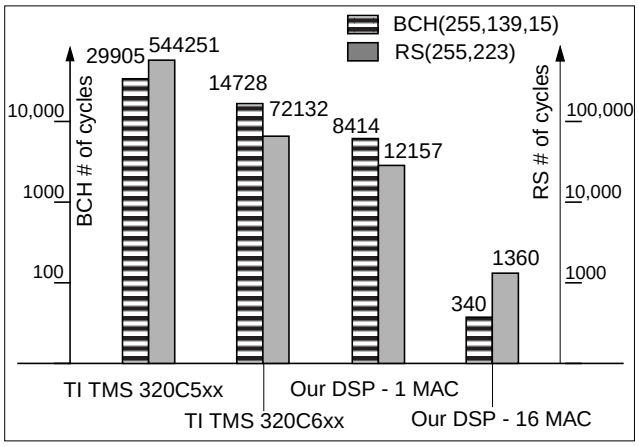

Fig. 5 Benchmarks/cycle counts for RS(255,223) and BCH(255,123) codes

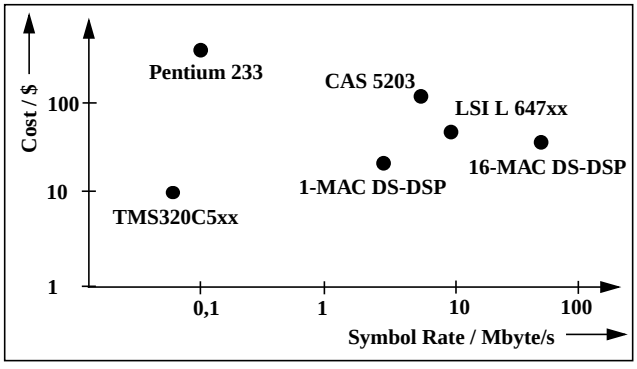

Fig. 6 Speed - cost product of different RS(255,223) decoder implementations, CAS 5203 and LSI L 647xx are commercial custom-built RS-decoder products

All cycle counts reflect mainly arithmetical operations an can increase due to memory transfer operations between the parts of the algorithm. In Fig. 6 a speed-cost-product diagram is depicted that assumes high volume production and estimated prices.

5. CONCLUSIONS

We have developed an architecture as well as specific datapath components for a DS-DSP tailored to perform encoder/decoder algorithms for block codes. Benchmarks and cost estimations prove the competitive position of our proposed architecture. Compared to a parallel architecture as TI's TMS320C6xx our parallel architecture employing 16 Galois-MACs is more than 50 times faster decoding a RS(255,233) code. With such a DS-DSP it should be possible to replace inflexible and expensive dedicated logic components in mobile communications and data storage devices while simultaneously having hardly any impact of existing software libraries.

REFERENCES

[1] G. Fettweis, „DSP Cores for Mobile Communications: Where are we going?,“ Proc. of ICASSP 1997, pp. 279-282.

[2] G. Fettweis, S. Wang, et al, „Strategies in a Cost Effective Implementation of the PDC Half-Rate Codec for Wireless Communications,“ IEEE 46th Veh. Tech. Conf., Atlanta, USA, pp. 203-207, 1996.

[3] I. Verbauwhede et al, „A low-power DSP engine for wireless communications,“ VLSI Signal Processing IX, IEEE, eds. W. Burleson et al, pp. 469-478, 1996.

[4] A. Abnous, J. Rabaey, „Ultra-Low Power Domain-Specific Multimedia Processors,“ VLSI Signal Processing IX, IEEE, eds. W. Burleson et al, pp. 459-468, 1996.

[5] M. Weiß, U. Walther, and G. Fettweis, „„A Structural Approach for Designing Performance Enhanced Signal Processors: A 1-MIPS GSM Fullrate Vocoder Case Study,“ Proc. of ICASSP 1997, pp. 4085-4088.

[6] W. Drescher and G. Fettweis, „VLSI Architectures for Multiplication in $GF(2^m)$ for Application Tailored Digital Signal Processors,“ VLSI Signal Processing IX, IEEE, eds. W. Burleson et al, 1996.

[7] W. Drescher, K. Bachmann, and G. Fettweis, „VLSI Architecture for Non-sequential Inversion over $GF(2^m)$ Using the Euclidean Algorithm,“ Proc. of ICSPAT 97, pp.631-634.

[8] A. G. Akritas, *Elements of Computer Algebra*, New York, NJ: Wiley, 1989.

[9] S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[10] R. T. Chien, „Cyclic Decoding Procedure for the Bose-Chaudhuri-Hocquenghem codes,“ *IEEE Transactions on Information Theory*, IT-11, pp. 549-557, Oct. 1965.