A Processor Architecture for Multi-Dimensional Parity Check Code Processing

Ryota ENDO , Hiroki OHSAWA*,
Keishi SAKANUSHI, Yoshinori TAKEUCHI, and Masaharu IMAI

Graduate School of Information Science and Technology Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871 JAPAN
{r-endou,sakanusi,takeuchi,imai}@ist.osaka-u.ac.jp

Abstract- Multi-Dimensional Parity Check (MDPC) code is an error correcting code which has been widely used for wireless communications under low error rate environment. In this study, a low-power processor for MDPC code processing is introduced and evaluated. Through experimental results, the proposed processor achieves about 90% lower in energy consumption compared with an implementation by a canonical RISC processor.

I. Introduction

Since size and battery capacitance of a small portable embedded system are limited, it is required that hardware in the system is small area and low power consumption. In recent years, with the advance of information technology, small portable embedded systems often have wireless communication. Therefore, in order to communicate in correct, error detecting and correcting method are important for small portable embedded systems. There is a method to detect and correct errors by using error correcting code [1]. Error correcting code adds redundant bits to input data, and corrects errors. For example, Multi-Dimensional Parity Check (MDPC) code [2] and extended hamming code [1] are widely used in wireless communicating systems.

Error correcting codes calculate code word from information symbol, and the code word is transmitted to recipient. Then, recipient receives code word and calculates error position information. Finally, detect and correct errors.

Generally, code generation units are designed to calculate code word in short time for each Error Check Code (ECC) and word length.

However, in MDPC code, code generation units for even a certain word length can calculate code word with different word length by calculating recursively.

Note that MDPC code includes a lot of bitwise computations. Since, a general purpose processor generally does not have bitwise operations, it spends many cycles and large power consumption to encode and decode.

One of idea to reduce energy consumption is to use a specific accelerator. However, MDPC implementation with specific accelerator also spends many execution cycles and energy because of the overheads on sending and receiving data between processor and specific accelerator[3].

[3] reports MDPC implementation with Application domain Specific Instruction set Processor (ASIP) is effective in encode, but experimental result in decode is not shown. In this paper, we design a MDPC implementation using ASIP, and evaluate effectiveness not only in encode but also in decode.

The rest of this paper is organized as follows. Section II introduces MDPC code. Then, proposed low-power processor is introduced in section III. Section IV shows experimental result. Finally, section V concludes this paper and indicates future problem.

II. Error Correcting Code

This section introduces encoding and decoding in MDPC code.

MDPC code adds AM bit into A^M bit information symbol. MDPC code can corrects one bit error and detects two bit errors where $M \geq 4$ [2].

A. Encode

Let A^M be the length of information symbol X, and let $\left\{u_{i_1,\ldots,i_M}, i_j \in \{1,\ldots,A\}\right\}$ be components of X. In this case, MDPC code $p_{m,j}$ is calculated as follows.

$$p_{m,j} = \sum_{i_1,\ldots,i_{m-1}} \sum_{i_{m+1},\ldots,i_M} u_{(i_1,\ldots,i_{m-1}),j,(i_{m+1},\ldots,i_M)} \quad (1)$$

Where $m = 1,\ldots,M$ and $j = 1,\ldots,A$. Note that additions in this equation are bit exclusive OR operations.

B. Recursiveness in MDPC code

Since MDPC code has recursiveness, code word is easily generated. This paragraph shows an example of the recursiveness where $A = 2$ and $M = 2$ or $M = 3$.

*At present, Fuji Xerox Co., Ltd

data	[4] $u_{2,2}$	[3] $u_{2,1}$	[2] $u_{1,2}$	[1] $u_{1,1}$
$p_{1,1}$			⊕	⊕
$p_{1,2}$	⊕	⊕		
$p_{2,1}$		⊕		⊕
$p_{2,2}$	⊕		⊕	

Fig. 1. MDPC code computing where $A = 2, M = 2$

data	[8] $u_{2,2,2}$	[7] $u_{2,2,1}$	[6] $u_{2,1,2}$	[5] $u_{2,1,1}$	[4] $u_{1,1,2}$	[3] $u_{1,2,1}$	[2] $u_{1,1,2}$	[1] $u_{1,1,1}$
$p_{1,1}$					⊕	⊕	⊕	⊕
$p_{1,2}$	⊕	⊕	⊕	⊕				
$p_{2,1}$			⊕	⊕			⊕	⊕
$p_{2,2}$	⊕	⊕			⊕	⊕		
$p_{3,1}$		⊕		⊕		⊕		⊕
$p_{3,2}$	⊕		⊕		⊕		⊕	

Fig. 2. MDPC code computing where $A = 2, M = 3$

MDPC code with $A = 2, M = 2$ is computed as follows.

$$p_{1,1} = \sum_{i_2} u_{1,i_2} = u_{1,2} \oplus u_{1,1} \tag{2}$$

$$p_{1,2} = \sum_{i_2} u_{2,i_2} = u_{2,2} \oplus u_{2,1} \tag{3}$$

$$p_{2,1} = \sum_{i_1} u_{i_1,1} = u_{2,1} \oplus u_{1,1} \tag{4}$$

$$p_{2,2} = \sum_{i_1} u_{i_1,2} = u_{2,2} \oplus u_{1,2} \tag{5}$$

Fig. 1 shows relations between MDPC code $p_{m,j}$ and information symbol X. In Fig. 1, $u_{1,1}, \ldots, u_{2,2}$ indicate information symbol X and $p_{1,1}, \ldots, p_{2,2}$ are MDPC code $p_{m,j}$ added to information symbol X. Moreover, $\oplus$ represents information symbol $u_{1,1}, \ldots, u_{2,2}$ which are required to compute $P_{m,j}$.

MDPC code with $A = 2$ and $M = 3$ is computed as follows by eq. (1).

$$p_{1,1} = \sum_{i_2,i_3} u_{1,i_2,i_3} = u_{1,2,2} \oplus u_{1,2,1} \oplus u_{1,1,2} \oplus u_{1,1,1} \tag{6}$$

$$p_{1,2} = \sum_{i_2,i_3} u_{2,i_2,i_3} = u_{2,2,2} \oplus u_{2,2,1} \oplus u_{2,1,2} \oplus u_{2,1,1} \tag{7}$$

$$p_{2,1} = \sum_{i_1} \sum_{i_3} u_{i_1,1,i_3} = u_{2,1,2} \oplus u_{2,1,1} \oplus u_{1,1,2} \oplus u_{1,1,1} \tag{8}$$

$$p_{2,2} = \sum_{i_1} \sum_{i_3} u_{i_1,2,i_3} = u_{2,2,2} \oplus u_{2,2,1} \oplus u_{1,2,2} \oplus u_{1,2,1} \tag{9}$$

$$p_{3,1} = \sum_{i_1,i_2} u_{i_1,i_2,1} = u_{2,2,1} \oplus u_{2,1,1} \oplus u_{1,2,1} \oplus u_{1,1,1} \tag{10}$$

$$p_{3,2} = \sum_{i_1,i_2} u_{i_1,i_2,2} = u_{2,2,2} \oplus u_{2,1,2} \oplus u_{1,2,2} \oplus u_{1,1,2} \tag{11}$$

Fig. 2 shows relations between MDPC code $p_{m,j}$ and information symbol X where $A = 2$ and $M = 3$.

Parts of computing patterns of $A = 2, M = 3$ (enclosed by dotted square in Fig. 2) are the same as computing patterns of $A = 2, M = 2$ (Fig. 1). This fact means that specific hardware for MDPC code with $M = n$ can be used as the calculating MDPC code with $M = n + 1$.

C. Decode

A code word which is transmitted by sender is denoted by Y and a code word received by recipient is denoted by Y'. In order to detect or correct errors in Y', comparison between MDPC code in received code word $p_{m,j}$ and MDPC code which is computed from Y', denoted by $p'_{m,j}$, is required.

The same specific hardware that used in encoding can be used for decoding because computing MDPC code from received code word Y' in decoding.

An example of decoding MDPC code where $A = 2$, $M = 4$ is shown in the below. In this case, $m = 1, \ldots, 4$, and $j = 1, 2$. Furthermore, calculate exclusive OR between $p_{m,j}$ and $p'_{m,j}$, and the block which $j = 1$ is denoted by $p'_{p_m,1}$, $j = 2$ is denoted by $p'_{p_m,2}$

1. $p'_{m,j}$ XOR $p_{m,j}$ = "00000000"
   In the case that result of exclusive OR between $p_{m,j}$ and $p'_{m,j}$ equals to 0, there is no error in the received code word Y'.

2. $p'_{p_m,1}$ XOR $p'_{p_m,2}$ = "1111"
   In the case that for all m, either $j = 1$ or $j = 2$ bit in all bits of $p'_{m,j}$ is inverted MDPC code $p_{m,j}$, there is one error in the received code word Y'. Moreover, the inverted bits in $j = 2$ indicate the position where error occurred. For instance, there is an error at the third bit in the information symbol part in the received code word Y', result of $p'_{m,j}$ XOR $p_{m,j}$ indicates 3 in binary as follows.
   $$p'_{4,2}, \ldots, p'_{1,2} = \text{"0011"} \tag{12}$$

3. $p'_{p_m,1}$ XOR $p'_{p_m,2}$ = "0001", "0010", "0100", *or* "1000"
   In the case that for the all m in $p'_{m,j}$, only one bit either $j = 1$ or $j = 2$ is inverted, there is one error in the MDPC code part of the received code word Y'. In this case, error correction is not be required because no error in information symbol in the received code word Y'.

4. Not included in any cases in the above, there are errors more than two bits in the received code word.

TABLE I
INSTRUCTION SET OF BROWNIE MICRO 16

Operation class	Operation
Arithmetic Op	ADD, SUB, Arith shift
Logical Op	AND, OR, XOR, NOT, Logical shift
Comparison Op	Equal, Not equal, Unsigned Comp
Immediate Op	ADD, Logical shift, Load Immediate
Bit Op	Bit set, clear bit, Bit test
Load/Store Op	Load half word, Store half word
Branch Op	Branch
Jump Op	Immediate jump, Indirect jump
Interrupt Op	TRAP, RETI
Special Op	NOP, SLEEP

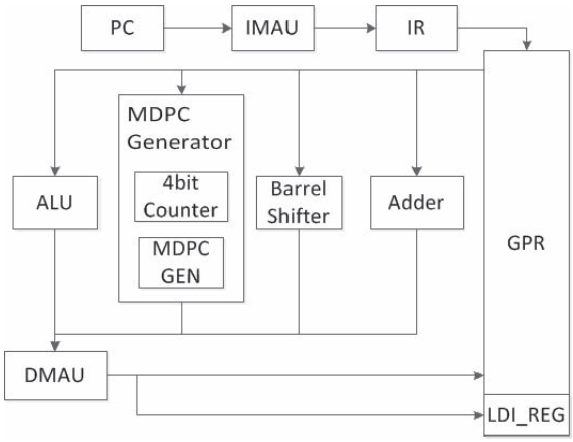

Fig. 3. Proposed Processor Architecture

III. PROPOSED PROCESSOR ARCHITECTURE FOR MDPC CODE

This section introduces a processor architecture specified for MDPC code processing. The proposed processor is designed by adding specific instructions and hardware to a processor, Brownie Micro 16 provided by ASIP Solutions, Inc. We designed the proposed processor using application domain specific instruction set processor development system, ASIP Meister [4] [5] [6]. Table I shows instruction set of Brownie Micro 16, and Fig. 3 shows the proposed processor architecture.

A. Custom Instructions for MDPC Code

We added custom instructions for MDPC code to Brownie Micro 16. This section explains the behavior of the proposed instructions.

MDPCGEN (MDPC Generate)
: MDPCGEN operation computes MDPC code on 16 bit specific hardware and 4 bit counter that holds word number x. By repeating this instruction n times, $16 \times n$ bit MDPC code is computed. Two General purpose registers, denoted by *rd* and *rs*, are operand fields for this operation. The x-th word is stored to *rs* and a result of $x-1$ times repeated MDPCGEN operations stored to *rd*. First, compute MDPC code using an information symbol in *rs*, and aggregate with MDPC code in *rd*. Second, compute 16 bit parity of the information symbol in the *rs*, then aggregate with MDPC code in *rd* according to the value of the 4 bit counter.

MDPCCHK (MDPC Check)
: MDPCCHK operation identifies that MDPC code in the received code word $p_{m,j}$ includes error or not and computes error position. Two General purpose registers, denoted by *rd* and *rs*, are operand fields for this operation. Received MDPC code $p_{m,j}$ is stored to *rd* and computed MDPC code $p'_{m,j}$, which is computed MDPC code from Y', is stored to *rs*. This operation creates the 14th and 15th bits of *rd* which mean type of error. The 14th and 15th bits equal "00" where there are no errors, "01" where there is one error in the part of information symbol of received code word Y', "10" where there is one error in the part of computed MDPC code $p'_{m,j}$, "11" where there are more than two errors. Moreover, the 0th ... 7th bits indicate the position where error occurred.

MDPCFIX (MDPC FIX)
: MDPCFIX operation corrects one bit error according to the result of MDPCCHK operation. Two General purpose registers, denoted by *rd* and *rs*, are operand fields for this operation. A 16 bit word which involves one bit error is stored to *rd* and the result of MDPCCHK operation is stored to *rs*. The 0th ... 3rd bits in *rs* indicate error position of the 16 bit word stored to *rd*. This operation inverts one bit in *rd* according to error information in *rs*. Before executing this operation, the word which involves one bit error must be identified by RISC instructions according to the result of MDPCCHK operation.

MDPCINIT (MDPC Initialize)
: MDPCINIT operation initializes 4 bit counter for MDPCGEN operation. This operation must be executed before MDPCGEN operation.

LDI (Load and Increment)
: LDI operation increments specific register *LDIREG* and loads data from a memory according to the value of *LDIREG*. Operand field for this operation is a general purpose register, which is denoted by *rd*. This operation loads data from a memory and stores to *rd*. This operation can decrease execution cycles since both address increment and data load are executed at once in the situation that data stored to consecutive addresses.

LDIST (LDI register Store)
: LDIST operation sets an initial setting to specific register *LDIREG*. In order to execute this operation, an address required to be stored to *LDIREG* sets to operand field.

TABLE II
AREA AND POWER OF THE PROCESSORS

	Area[μm^2]	Power[$\mu W/MHz$]
BM-16	61,398	40.95
ASIP	72,532 (+18.1%)	46.59 (+13.8%)

TABLE III
MAX FREQUENCY OF THE PROCESSORS

	Max frequency[MHz]
BM-16	201.6
ASIP	188.7 (-6.4%)

DBRNZ (Decrement and Branch if Non Zero)
DBRNZ operation decrements general purpose register *GPR*5 and does indirect jump by the value of *GPR*5. Sign-extended operand *const* is added to PC if the value of *GPR*5 is not equals 0. This operation can decrement, determine loop counter, and do indirect jump at once.

IV. EXPERIMENTS

This section introduces experimental result for evaluating the proposed ASIP for MDPC code.

A. How to Experiment

We evaluated the proposed ASIP by comparing with Brownie Micro 16. In the experiment, execution cycle, energy consumption, area, power, and maximum frequency were compared. Operation voltage and frequency of both processors are $1.8V$ and $100MHz$. In order to measure execution cycle, we simulated programs which encode and decode 16, 32, 64, 128, and 256 bit information symbol on each processor.

In this paper, we use MDPC code with $A = 2$ because information symbol consists of multiple of 16 and Brownie Micro 16 is a 16 bit processor. In this case, information symbol is 2^m bit, and multiple of 16 is involved in 2^m.

B. Experimental Result

1. *Area, Power Consumption, Maximum Frequency*
Table II and III show comparison on area, power, and max frequency with Brownie Micro 16 and proposed ASIP. Furthermore, numbers in parenthesis of Table II and III indicate percentage of increase or decrease compared with Brownie Micro 16.

Area was estimated from RTL description by logic synthesis using Synopsys's Design Compiler and $0.18\mu m$ CMOS library. Constraint of logic synthesis was minimum area and its frequency was set to $100MHz$. Power consumption was estimated by switching information of circuit obtained from gate level simulation of repeating to encode and decode MDPC code using Mentor's ModelSim.

Compared with Brownie Micro 16, proposed ASIP is 18.1% larger in area and 13.8% upper in power consumption because MDPC code specific hardware and 4 bit counter for MDPCGEN instruction, and 16 bit register for LDI instruction are added.

TABLE IV
EXECUTION CYCLE FOR ENCODE [*cycles*]

input data [*bit*]	BM-16	ASIP
16	115	20 (-82.6%)
32	160	26 (-83.8%)
64	248	38 (-84.7%)
128	406	62 (-84.7%)
256	772	110 (-85.8%)

TABLE V
EXECUTION CYCLE FOR DECODE [*cycles*]

input data [*bit*]	BM-16	ASIP
16	157	36 (-77.1%)
32	202	42 (-79.2%)
64	290	54 (-81.4%)
128	448	78 (-82.6%)
256	814	126 (-84.5%)

2. *Execution Cycle*
Table IV shows execution cycle for encoding MDPC code 16, 32, 64, 128, and 256 bit, and Table V shows execution cycle for decoding and fixing one bit error.

Numbers in parenthesis of Table IV and Table V indicate percentage of decreasing compared with implementation only by RISC instructions.

By using custom instructions, execution cycle is decreased 82.6% to 85.8% in encode and 77.1% to 84.5% in decode.

3. *Energy Consumption*
Energy consumption is computed from execution cycle, power consumption, and frequency of operation. Let $N[cycles]$ be execution cycle for encoding and decoding, $P[\mu W]$ be power consumption, and $f[MHz]$ be frequency of operation. By using these symbols, energy consumption $E[nJ]$ can be computed as follows.

$$E = PN\frac{1}{f} \tag{13}$$

TABLE VI
ENERGY FOR ENCODE [nJ]

input data[bit]	BM-16	ASIP
16	4.83	0.93 (-80.7%)
32	6.67	1.21 (-81.9%)
64	10.16	1.77 (-82.6%)
128	16.63	2.89 (-82.6%)
256	31.61	5.12 (-83.8%)

TABLE VII
ENERGY FOR DECODE WITH ONE BIT ERROR CORRECTION [nJ]

input data[bit]	BM-16	ASIP
16	6.55	1.68 (-74.4%)
32	8.39	1.96 (-76.7%)
64	11.88	2.52 (-78.8%)
128	18.35	3.63 (-80.2%)
256	33.33	5.87 (-82.4%)

In eq. (13), $\frac{1}{f}$ indicates clock cycle, and $\frac{1}{f} = 1.0 \times 10^{-8}[s]$ because $f = 100[MHz]$ in this experiment.

Table VI, VII, and VIII show energy consumption for encoding or decoding 16, 32, 64, 128, and 256 bit information symbol. Energy consumption for encoding is shown in the Table VI, for decoding with one bit error correcting is shown in the Table VII, and for decoding without error correction is shown in the Table VIII.

Numbers in parenthesis of Table VI, VII, and VIII indicate percentage of decreasing compared with Brownie Micro 16.

By using the proposed custom instructions, energy consumption 80.7% to 83.8% in encode process, 74.4% to 82.4% in decode process with one bit error correction, and 76.7% to 83.1% in decode process without error correction.

C. Discussion

Note that the proposed processor is superior to Brownie Micro 16 in terms of energy consumption, while power consumption is inferior, because energy consumption is more affected by execution cycle than power consumption.

The overhead in area, power and max frequency of the proposed processor are inferior to that of Brownie Micro 16, but the difference is not critical for small portable embedded systems.

According to these experimental results, energy consumption for encoding and decoding decreased significantly by designing processor with specific instructions for MDPC code.

TABLE VIII
ENERGY FOR DECODE WITHOUT ERROR CORRECTION [nJ]

input data[bit]	BM-16	ASIP
16	5.41	1.26 (-76.7%)
32	7.25	1.54 (-78.8%)
64	10.73	2.10 (-80.5%)
128	17.20	3.21 (-81.3%)
256	32.19	5.45 (-83.1%)

V. CONCLUSIONS

This paper introduced a processor architecture specified for MDPC code processing. The proposed custom instructions for MDPC code decrease execution cycles and energy consumption for encoding and decoding.

In future, we will develop an accelerator for MDPC code and add experimental comparison using the accelerator.

ACKNOWLEDGMENT

This study is partly supported by the collaborative research project titled "Development of Health care Devices and Systems for Ubiquitous Bioinstrumentation".

REFERENCES

[1] R.W, Hamming, "Error detecting and error correcting codes," Bell System Technical Journal, Vol. 29, No. 2, pp. 147 – 160, 1950.

[2] T. F. Wong and J. M. Shea, "Multi-Dimensional Parity-Check Codes for Bursty Channels," in Proceedings of 2001 IEEE International Symposium Information Theory, Washington, D.C., p.123, 2001.

[3] Yoshinori Takeuchi, Hiroki Ohsawa, Tomohiro Kondo, Hirofumi Iwato, Keishi Sakanushi, and Masaharu Imai, "Low Energy MDPC Implementation using Special Instructions on Application Domain Specific Instruction-Set Processor," in Proceedings of Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ACS 2010), pp. 47 – 52 , 2010.

[4] Masaharu Imai "Synthesis of Application Specific CPU Core," Proceedings of the Synthesis and Simulation Meeting and International Exchange (SASIMI), V-I, 1989.

[5] Makiko Itoh, Shigeaki Higaki, Jun Sato, Akichika Shiomi, Yoshinori Takeuchi, Akira Kitajima, and Masaharu Imai: PEAS-III: An ASIP Design Environment, Proceedings of International Conference on Computer Design: VLSI in Computers and Processors (ICCD), pp. 430-436, September 2000.

[6] Yuki Kobayashi, Yoshinori Takeuchi, and Masaharu Imai: ASIP Meister, Processor Description Languages, Prabhat Mishra and Nikil Dutt Eds, Elsevier, 2008, pp. 163-182, 2008.