Fast Parallel CRC Algorithm and Implementation on a Configurable Processor

H. Michael Ji, and Earl Killian
Tensilica, Inc.
3255-6 Scott Blvd
Santa Clara, CA 95054

***Abstract*--In this paper we present a fast cyclic redundancy check (CRC) algorithm that performs CRC computation for any length of message in parallel. For a given message with any length, we first chunk the message into blocks, each of which has a fixed size equal to the degree of the generator polynomial. Then we perform CRC computation among the chunked blocks in parallel using Galois Field multiplication and accumulation (GFMAC). Theoretically our fast parallel CRC algorithm can achieve unlimited speedup over the bit-serial algorithm or byte-wise table lookup algorithm at the expense of adding enough GFMAC units. Our algorithm can perform CRC computation for any lengthy message with 2 to 3 clock cycles. In practice, we choose to use a configurable processor where a customized instruction is added to perform multiple pairs of GF multiplication and accumulation. For example, a 4-GFMAC implementation can compute a 32-bit CRC in 2 to 3 cycles for a 16-byte message. This level of performance is hundreds of times faster than bit-serial CRC algorithm or tens of times faster than byte-wise parallel CRC algorithm. The generator polynomial can be chosen to be software programmed or hard-coded. Our algorithm adds only a small number of logical gates to the processor core.**

I. INTRODUCTION

Cyclic redundancy check (CRC) is popularly used to detect errors in digital transmission and storage areas. A frame check sequence is appended to the original message for error detection. CRC has been effectively employed in many communication protocols such as Ethernet, fiber distributed data interface (FDDI), asynchronous transfer mode (ATM), and various digital subscriber line technologies such as ADSL/VDSL, etc. CRC calculation can be performed in hardware or software. The common hardware solution is the linear feedback shift register (LFSR), which is a simple bit-serial architecture for both encoding and decoding the message. This approach typically calculates the CRC for a N-bit message in N clock cycles. This approach is not efficient at high bit rates. The basic bit-serial algorithm can be accelerated by parallel processing a number of bits (e.g. a byte) [8]. The most popular algorithms are using table lookup [8][9]. A software implementation is given in [3] using shift-and-add operations. However, CRC is more commonly computed in hardware. A parallel hardware solution is first presented for particular generator polynomials in [6]. The bit-serial LFSR algorithm is treated as an M-tap finite impulse response filter using z-transformation in [1], which gives the theoretical basis for parallel CRC calculation. A parallel CRC circuit is derived from the LFSR state transition equation for a given generator polynomial [7] where a number of shift-and-subtract operations are merged to process M bits where M is equal to the number of bits in the CRC. These proposed methods are either exemplary or involve complicated algebraic transformations. The speedups over the bit-serial algorithm obtained in all these algorithms are about a factor of M. In this paper we present a fast CRC algorithm to process multiple sets of M bits in parallel. In other words, our CRC algorithm can be many times faster than these parallel M-bit algorithms or table lookup approach.

In [10] an approach is presented for parallel CRC by using LFSR cascading where automatic circuit is generated using VHDL description language. A nibble-wise parallel CRC algorithm is presented in [4] which yields multiple times of speedup over the standard table lookup algorithms. Joshi et al [4] also present an algorithm with speedups $>M$ but focused on the 32-bit CRC computed on a general purpose PowerPC microprocessor with AltiVec technology using that instruction set's permutation capability as a table lookup. Their approach has significant overhead.

In this paper we uncover the underlying theory for parallel CRC based on Galois Field arithmetic. A novel parallel CRC algorithm is presented which performs CRC computation for a N-bit message in O(log$M$) levels of logic (often 2 to 3 cycles) using N/M parallel GF multipliers whose results are combined using $\log_2(N/M)$ levels of XOR. For variable-length V-bit messages, N bits are processed per cycle (the extra cycle latency can be hidden by taking advantage of instruction-level parallelism) with a time of O(V/N) cycles. The algorithm is implemented on a configurable processor where the generator polynomial (with any degree) can be programmed by software. The rest of this paper is organized as follows: we present our fast parallel CRC algorithm in Section 2; its implementation on a configurable processor [11][12] is given in Section 3; and then we conclude in Section 4.

II. FAST PARALLEL CRC ALGORITHM

A CRC is the remainder after dividing the original message concatenated with M zero bits by a generator polynomial in

binary modulo-2 arithmetic. As said earlier, common CRC algorithms are bit-serial and byte-serial with table lookup. In the bit-serial CRC algorithm, 1 input bit is processed at a time as the name indicates, using long division. First we append *M* zeros (*M* is the number of bits in CRC) to the original *N*-bit message to the least significant bit (LSB) end and define the appended message as a running message. Second, check the most significant bit (MSB) of the running message. If the MSB of the running message is 1, subtract[1] the generator polynomial from the *M* most significant bits of this running message and shift the result to left by 1 bit and store the result to the running message. Otherwise, (i.e., the MSB of the running message is 0), just shift the running message to left by 1 bit and store the result to the running message. The second step is repeated *N* times and at the end the remainder will be the *M* most significant bits, which is the CRC. Note that in bit-serial CRC algorithm, we need to perform *N* times of operation where in each operation we need to perform checking the MSB bit, modulo-2 subtraction (conditionally), and left shift.

In byte-wise CRC algorithm, one byte is checked at a time. In the first step, similar to bit-serial CRC algorithm, we append *M* zeros to the original message after the LSB. If the original message size *N* is not a multiple of 8, we need to pre-append a number of 0's (in the range of 1 to 7) before the MSB to make the appended message having size of multiple of 8. We define this appended message as a running message. Second, perform table lookup based on the MSB 8 bits to find the *M*-bit remainder. This *M*-bit remainder will be XORed with the following MSB byte in the running message. Then we left shift the running message by 8 bits. Repeat the second step by $\lceil N/8 \rceil$ times. In total there are $\lceil N/8 \rceil$ operations with $2^8 * M$ bit of memory for table lookup. Note that the table has 256 entries each of which has *M* bits. The table entries can be pre-computed since they only depend on the generator polynomial.

For a message with *N* bits, we can represent it in a binary form $[a_{N-1}a_{N-2}...a_0]$ where a_{N-1} denotes the MSB and a_0 the LSB. The original message can also be represented in a polynomial format:

$$A(x) = a_{N-1}x^{N-1} + a_{N-2}x^{N-2} + ... + a_0$$

For CRC computation, it is always associated with a generator polynomial. Let us denote the generator polynomial by binary representation $[g_M \; g_{M-1}...g_0]$ or equivalent polynomial format:

$$G(x) = g_M x^M + g_{M-1}x^{M-1} + ... + g_0$$

[1] Note that subtract is based on modulo-2 arithmetic. Also note that in modulo-2 subtract is equivalent to add which is the same as exclusive-or (i.e., XOR) operation.

where *M* is the degree of the generator polynomial, g_M is the MSB and g_0 is the LSB. Note that the binary representation of the generator polynomial has *M+1* bits. Since the MSB is always 1, only *M* bits $[g_{M-1}...g_0]$ need to be set.

Given the original message *A(x)* and generator polynomial *G(x)*, we can compute the CRC by appending *M* zeros after the LSB and then dividing the appended message by *G(x)*. Equivalently, it is:

$$CRC[A(x)] = (A(x)x^M) \bmod G(x) \quad (1)$$

Note that the arithmetic in CRC calculation is modulo-2 for easy computation.

After CRC is computed, the CRC is appended to the original message for transmission or storage. Before we present our fast parallel CRC algorithm, we need the following lemma, which gives the well-known congruence properties for polynomial division.

Lemma 1. Congruence Properties

$$(A(x) + B(x)) \bmod G(x) = A(x) \bmod G(x) + B(x) \bmod G(x)$$

$$A(x)B(x) \bmod G(x) = (A(x) \bmod G(x))\,(B(x) \bmod G(x))$$

Proof: See [1].

There is a system of arithmetic known as Galois field arithmetic. A Galois field is an example of algebraic field, which has a set of elements called numbers or field elements, and a definition of two operations called addition and multiplication such that the formal properties satisfied by the real arithmetic system (e.g. commutativity, associativity, and distributivity) are satisfied. For each positive integer *M*, there is a Galois field called $GF(2^M)$ that has 2^M elements in it. The set of elements of $GF(2^M)$ can be represented as the set of *M*-bit binary numbers in the range from 0 to 2^M-1. The addition over Galois Field $GF(2^M)$ is defined as bit-by-bit modulo-2 addition which is equivalent to exclusive-or (i.e., XOR) operation. Galois Field multiplication is more complicated having the structure of shift and exclusive-or. The generator polynomial *G(x)* is irreducible over *GF(2)* of degree *M*, i.e., *G(x)* can have only coefficients equal to 0 or 1, and can not be factored into the product of two such polynomials over *GF(2)*. Multiplication of two elements *a* and *b* in $GF(2^M)$ will produce an element *c=ab* by polynomial multiplication modulo the irreducible polynomial *G(x)*, i.e.,

$$C(x) = A(x)B(x) \bmod G(x) \quad (2)$$

where *A(x)*, *B(x)* and *C(x)* is the polynomial form of *a*, *b*, and *c*, respectively. Since the coefficients are added and multiplied by the bit operations of *GF(2)*, the polynomial product is equivalent to the shift and exclusive-or operations. The modulo *G(x)* operations specifies the rule for folding the overflow bits back into the *M* bits of the finite field element. Thus the product of two elements in $GF(2^M)$ also produces another element of $GF(2^M)$, i.e., the word length does not increase (still *M* bits).

In our fast parallel CRC algorithm, first we need to chunk the original message into blocks with each of the blocks having *M* bits. Without loss of generality, we assume that the number of bits *N* is divisible by *M*, i.e., *N=nM*, where *n* is an integer.[2] Thus we have:

$$A(x) = (a_{nM-1}x^{M-1} + ... + a_{(n-1)M})x^{(n-1)M} + ... + (a_{M-1}x^{M-1} + ... + a_1 x + a_0) \quad (3)$$
$$= W_{n-1}(x)x^{(n-1)M} + ... + W_1(x)x^M + W_0(x)$$

where $W_i(x)$ is associated with the *i*-th block given by:

$$W_i(x) = a_{(i+1)M-1}X^{M-1} + ... + a_{iM}$$

From equation (1) and the congruence property, we can compute CRC for the chunked message (3) by:

$$CRC[A(x)] = W_{n-1}x^{nM} \bmod G(x) + ... + W_0 x^M \bmod G(x)$$

Furthermore we have:

$$W_i(x)x^{(i+1)M} \bmod G(x) = W_i(x) \bmod G(x)\, x^{(i+1)M} \bmod G(x)$$

for *i = 0, 0, …, n-1*. Note that the degree of the polynomial $W_i(x)$ for each chunk is less than *M*. Then the modulo of $W_i(x)$ by *G(x)* is $W_i(x)$, i.e., $W_i(x) \bmod G(x) = W_i(x)$.

Let us define coefficients, $b_i = x^{(i+1)M} \bmod G(x)$ for *i=0, 1, …, n-1*. Note that b_i is determined only by the generator polynomial and is independent of the message. We can pre-compute these coefficients and store them in memory or registers. Therefore we have:

$$CRC[A(x)] = W_{n-1} \otimes b_{n-1} \oplus ... \oplus W_0 \otimes b_0 \quad (4)$$

Note that the operations $\otimes$, $\oplus$ in the above equation is Galois Field multiplication, addition (or accumulation) over $GF(2^M)$, respectively. From the above equation, we present our fast parallel CRC algorithm as follows:

Fast Parallel CRC Algorithm:

1. Load *N*-bit message in the chunked form of $(W_{n-1}W_{n-2}...W_0)$ where each chunk is *M* bits (Note: *N = nM*).
2. Initially setup the generator polynomial *G(x)* and its degree *M*. In the same time, pre-compute the beta factors $(b_{n-1}\, b_{n-2} ... b_0)$ which depend only on the polynomial and its degree.
3. Perform *n*-pair Galois field multiplications in parallel and then XOR the products. This generates the CRC results.

The above algorithm is illustrated in Figure 1.

[2] For a message with undividable length, we can pre-append the message with a small number of 0's at the MSB side to make it become divisible.

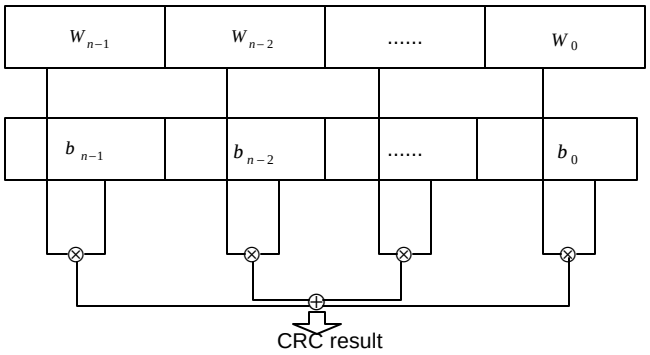

FIGURE 1 ILLUSTRATION OF FAST PARALLEL CRC ALGORITHM USING *N* GFMAC UNITS

A few remarks regarding the algorithm needs to be put in place:

- The size of each chunk (denoted by *c* in bits) can be chosen *M* or less. If *c* is chosen to be the number of bits in CRC, i.e., *c = M*, the original message will have *N/c = N/M = n* chunks. In this case, we need *n* pairs of GF multiplication and accumulation computation (GFMAC) units with each unit performing multiplication of *MxM* bits. If *M* is large, this operation of GF multipliers will have long latency. For example, if *M = 32*, this operation will be expected to have 3 cycle latency (to be shown in the next section). If *M = 8*, this will be expected to have 1 cycle latency. An appropriate parameter of *c* can be chosen so as to tradeoff between latency and hardware cost. Suppose *M* is 32. If we decide to chunk the original message with less than *M* bits each (say 8), this will require GF multiplier to multiply a pair of 8x32 bits which can be performed with a lower latency. However, this would require more GF multipliers to achieve the same level of performance as wider 32x32 GFMACs.
- As another example, if *N=128* and *M = 8*, this will require 16 GF 8x8 multipliers. If *N=128* and *M = 32*, this will require 4 GF 32x32 multipliers. In the rest of this paper, we assume that the message is chunked into *M* bits.
- The length of the original message *N* may be quite large. This will require a large number of GFMAC units. Suppose a limited number (denoted by *p*) of GFMAC units are implemented for lower hardware cost. Then we can compute the CRC by calling this operation, which does *p* pairs of GFMAC, $\lceil N/p \rceil$ times. For example, Horner's rule can be used to reduce a *N*-bit message to *p*M* bits by successive multiplication by $x^{p*M} \bmod G(x)$, and then the algorithm of Figure 1 is used to compute the final CRC.
- The beta factors $(b_{n-1}\, b_{n-2} ... b_0)$ do not depend on the incoming message but only on the generator polynomial. Since the polynomial is static, these

should be pre-computed once and used repeatedly in the CRC calculation.

- If the GFMAC units are used only for CRC calculation using a fixed generator polynomial, fixed circuits that multiply by b_i may be used to reduce hardware cost.
- To compute the beta factors, we have

$$b_0 = x^M \bmod G(x) = [g_{M-1} \dots g_0]$$
$$b_1 = x^{2M} \bmod G(x) = b_0 \otimes b_0 = b_0^2$$
$$\dots$$
$$b_{n-1} = x^{nM} \bmod G(x) = b_0^n$$

The above equation implies that b_0 is equal to the generator polynomial in binary representation without the MSB. All the other beta factors $(b_{n-1}\, b_{n-2} \dots b_1)$ can be derived from this base factor by *n, n-1, …, 2* times of GF multiplication, respectively.

III. Implementation and Performance Analysis

As mentioned earlier, Galois field multiplication of two elements consists of two steps:

(1) multiply the two elements by shift and exclusive-or operations to generate *2M-1* bit intermediate product;
(2) divide the intermediate product by the irreducible polynomial *G(x)* for folding the overflow bits back into the *M*-bit of the field element using shift and exclusive-or operations.

To be more specific, GF multiplication of *a* and *b* over $GF(2^M)$ can be described in Figure 2.[3] In the first loop, for each bit in *b* say at position *i*, if the bit is 1, we XOR with *a* shifted left by *i*; otherwise XOR with 0. This operation repeats *M* times to generate *2M-1* bit immediate product. In actual hardware implementation, these *M* steps can operate in parallel to generate the operand to be XORed together at the *(M+1)*-th step. For the second loop, we check the MSB of each running remainder. If it is 1, we XOR the running remainder with the generator polynomial left shifted by *(j-M)* times. Otherwise, we XOR with 0's. This operation is repeated *M-1* times, reducing the length of running remainder by 1-bit at each step. At the end, the remainder will have length of *M* bits, which gives the final product of GF multiplication. Note that in the second loop there is data dependency between one step of operation and its previous step. This implies that there will be *M-1* levels of logic in the second loop. In total by combining the first and second loops there are about *M* levels of MUX/XOR logic.

```
for i = 0,..., M −1
  product[2M − 2 : 0] ^= {b_i ? ([a_{M−1} ... a_0] << i) : (2M − 1)'b0};
// product has 2M −1 bits
// setup MSB to calculate remainder
rem[2M − 2] = product[2M − 2];
for j = 2M − 2,..., M
  // XOR with POLY if MSB is 1
  rem[ j − 1: 0] ^= {rem[ j] ? ([g_{M−1} ... g_0] << ( j − M)) : j'b0};
```

FIGURE 2 GF MULTIPLICATION OF A AND B OVER GF(2^M)

When the generator polynomial *G(x)* is fixed rather than programmable, a faster method can be used. For each multiplier bit *i* and multiplicand bit *j*, the contribution to the product is $a_i b_j\, x^{i+j} \bmod G(x)$. Note that $H(x) = x^{i+j} \bmod G(x)$ can be computed statically when *G(x)* is fixed, and the $a_i b_j$ term can be summed to each output position *k (*subject to *i+j=k)* for which h_k is 1.

Since GF multiplication is followed by addition in CRC calculation, we can integrate GF multipliers with adder such that multiply-accumulate (MAC) operation can be carried out in one instruction.

The GFMAC units can be implemented in hardware. The number of GFMAC units determines how much speedup we can achieve for CRC calculation over bit-serial approach. For *n* pairs of GFMAC units, this will give *n*M* times improvement over bit-serial approach. Theoretically we can achieve unlimited speedup by increasing *n*. In practice, more hardware cost will be introduced as *n* grows. In this paper, we choose to use a configurable processor like Xtensa that allows to adding customized instructions to accelerate computation-intensive operations [11][12]. Note that these customized instructions are associated with additional hardware logic added to the processor. We have developed the customized instructions using Tensilica Instruction Extension (TIE) language[4] to describe the GFMAC operations. We have implemented a Galois Field multiply and accumulation instruction called (GFMAC*M*x*M*P*n*) which does *n* pairs of *M*x*M* bits of GF multiply and then XOR the products. This enables to generate the CRC for *n*M*-bit message with a single instruction. We have used EDA tools to perform synthesis of this instruction. The number of gates and timing are reported in Table 1 using 0.18um process technology. Note that the reported timing does not include the time to setup the input and output wires. From this table, we estimate that GFMAC8x8, GFMAC32x32 can be fit in 1 and 3 cycles for a processor at 200 MHz, respectively. Also note that GFMAC with hard-coded polynomial yields smaller

[3] In the equation, we loosely use the notation of Verilog: *product[2M-2:0]* means that it has *2M-1* bits, *(2M-1)'b0* represents *2M-1* zeros.

[4] TIE is a high-level language that allows rapid definition of new instructions. TIE has similar style as a subset of Verilog hardware description language.

hardware cost and much better timing but lack of programmability, GFMAC with soft-coded polynomial produces larger hardware cost and worse timing but allows the generator polynomial to be programmed by software. A right choice can be made based on the underlying application's needs on whether the generator polynomial will change or not. Also note that as more GFMAC units are added, this will cause the hardware cost to increase linearly.

GFMAC Type	# of GFMAC Units	Hard/soft-coded Polynomial	Straightforward TIE	
			# Gates	*Timing (ns)*
GFMAC8x8	16	Hard-coded	3K	3.29
	16	Soft-coded	5K	3.37
GFMAC32x32	4	Hard-coded	21K	4.32
	4	Soft-coded	26K	12.25

TABLE 1 HARDWARE COST AND TIMING OF GFMAC UNITS @0.18 TECHNOLGY

Note that in the table for hard-coded polynomial, the EDA synthesis tool performs optimization to yield much smaller number of gates. For the hard-coded results reported in the table, the underlying primitive polynomial used for GFMAC8x8 and GFMAC32x32 is 0x1D in hex format (with degree of 8) [5], and 0x4C11DB7 in hex format (with degree of 32) [6], respectively. For different types of hard-coded polynomials, it will generate different hardware cost.[7] Also note that the results reported in the table are based on a straightforward TIE implementation. We have also implemented the instruction in an optimal way, which yields better timing for both hard-coded and soft-coded cases. The GFMAC units can also be used for other forward error correction (FEC) such as Reed-Solomon encoding/decoding [4].

The results in the table also shows that the CRC can be calculated within 2 to 3 cycles[8] for a 128-bit (i.e., 16 byte) message with a configurable processor at 200 MHz where 16 pairs of GFMAC8x8 units or 4 pairs of GFMAC32x32 units are added.

IV. PERSPECTIVE AND SUMMARY

We presented a fast cyclic redundancy check (CRC) algorithm that performs CRC computation for any length of message in parallel. For a given message with any length, we first chunk the message into blocks, each of which has a fixed size of *M*-bits equal to the number of bits in CRC. Then we perform CRC computation among the chunked blocks in parallel using Galois Field (GF) multipliers. Finally the products of GF multipliers are accumulated (i.e., XORed) to generate the final CRC. Theoretically, our fast parallel CRC algorithm can achieve unlimited speedup over the existing bit-serial or byte-wise table lookup CRC algorithm. Our algorithm can perform CRC computation for any lengthy message with 2 to 3 cycles at the expense of adding enough number of GFMAC units. In practice, we choose to use a configurable processor where a customized instruction is added to perform multiple pairs of GF multiplication and accumulation. This yields to compute CRC within 2 to 3 cycles for a 16-byte message. This level of performance is many times faster than bit-serial CRC algorithm and byte-wise table lookup CRC algorithm. Our algorithm requires the addition of a small number of logical gates for customized GFMAC instruction and registers to hold the beta factors. The generator polynomial can be either software programmed or hard-coded.

[5] Note that 0x1D hex represents the polynomial $x^8 + x^4 + x^3 + x^2 + 1$. The most significant bit (x^8) does not need to be represented since it is implicitly indicated by the polynomial degree of 8. This polynomial is popularly used in GSM channel coding and ADSL/VDSL error control.

[6] This polynomial with degree of 32 is popularly used in Ethernet.

[7] The actual hardware cost depends on the number of bits that are set to 1 in the generator polynomial.

[8] The exact number of cycles depends on whether primitive polynomial is soft-coded or hard-coded.

ACKNOWLEDGEMENT

We are grateful to our colleague, Himanshu Sanghavi, for stimulating discussion.

REFERENCES

[1] G. Albertengo and R. Sisto, Parallel CRC Generation, IEEE Micro, vol. 10, no. 5, Oct. 1990.

[2] E. Berlekamp, Algebraic Coding Theory, McGraw-Hill, 1968.

[3] D. C. Feldmeier, Fast Software Implementation of Error Correcting Codes, IEEE/ACM Trans. Networking, vol. 3, no. 6, Dec. 1995.

[4] H. Ji, An optimized processor for fast Reed-Solomon encoding and decoding, Proc. of IEEE Int. Conf. On Acoustics Speech and Signal Processing (ICASSP), Orlando, FL, May, 2002.

[5] S. Joshi, P. Dubey and M. Kaplan, A New Parallel Algorithm for CRC Generation, IEEE ICC'2000, New Orleans, Louisiana, June 2000.

[6] A. K. Pandeya and T. J. Cassa, Parallel CRC Lets Many Lines Use One Circuit, Computer Design, vol. 14, no. 9, Sept. 1975.

[7] T. B. Pei and C. Zukowski, High-Speed Parallel CRC Circuits in VLSI, IEEE Trans. Comm., vol. 40, no. 4, April 1992.

[8] A. Perez, Byte-wise CRC Calculations, IEEE MICRO, June 1983.

[9] D. V. Sarwate, Computation of Cyclic Redundancy Checks via Table Lookup, Communications of the ACM, vol. 31, no. 8, August 1988.

[10] M. Sprachmann, Automatic Generation of Parallel CRC Circuits, IEEE Design & Test of Computers, vol. 18, no. 3, May 2001.

[11] Tensilica, Inc., http://www.tensilica.com.

[12] A. Wang, E. Killian, D. Maydan and C. Rowen, Hardware/Software Instruction Set Configurability for System-on-Chip Processors, Proc. of the 38-th Design Automation Conference, Las Vegas, NV, June 2001.