Technica Report CS75015-R

SOME CLASSICAL MATHEMATICAL RESULTS RELATED TO THE PROBLEMS OF THE FIRMWARE/HARDWARE INTERFACE

T. C. Wesselkamper

July 1975

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Some Classical Mathematical Results Related to the Problems of the Firmware/Hardware Interface

by

T. C. Wesselkamper

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia

Abstract

The paper reviews the Shannon and Reed-Muller Decomposition Theorems and notes the relationship of the former to machine instruction set. It hypothesizes an instruction set based on Galois field operations and applies the divided difference methods of Newton to the automatic generation of a polynomial representation of an arbitrary function. Some numeric results are given for the fields GF(9) and GF(16). An extension of the methods to the representation of functions by rational forms is suggested.

Some Classical Mathematical Results Related to the Problems of the Firmware/Hardware Interface

by

T. C. Wesselkamper
Dept. of Computer Science
Virginia Polytechnic Institute
and State University
Blacksburg, Virginia

I. Background

If in 1975 one surveys instruction sets for digital computers of the present and past one finds a great similarity. A typical recent example is the Weisbecker machine [1]. In an excellent paper Joe Weisbecker "describes a simplified microcomputer architecture that offers maximum flexibility at minimum cost." [1, p. 41] We are told:

> "The ALU is an 8-bit logic network for performing binary subtract, logical 'and', 'or' , and 'exclusive or' on two 8-bit operands. One operand is the bus byte and the other is contained in the D register. The D register can also be shifted right one bit position. Add, subtract, and shift operations set a one bit overflow register...which can be tested by a branch instruction." [1, p. 43]

No attempt is made to explain this choice of functions. They provide a typical example of the operations provided by designers.

The second recent example is a description of HALL [2] , an assembler level language for the HYRMAN hardware simulator [3]. The HALL machine possesses nine arithmetic functions (binary and decimal addition and subtraction, 'and', 'or', 'exclusive or', left shift and right shift) and fourteen status instructions.

The work reported herein was supported in part by National Science Foundation Grant No. DCR74-18108.

These are given in Table I, below. (It should be noted that HALL does decimal arithmetic in a fashion analogous to the old IBM 1620 or to the S/360-370 "packed decimal" arithmetic.)

That these designs have not varied significantly in thirty years may be seen by comparing them to the instructions proposed by John von Neumann for the EDVAC machine in 1945 [4]. See Table II, below. This in spite of the dual facts that electronics can support much more varied design and that the class of problems to which computers have come to be applied is far wider than was envisioned when the first computers were designed for numeric work.

Certain characteristics may be noted as common to the instruction sets of all current machines:

1.) all provide arithmetic functions (addition, subtraction, multiplication) modulo 2^L or 2^L-1, where L is the word size in bits of the machine;

2.) all provide bit-wise logic operations (conjunction, disjunction, non-equivalence).

The first of these characteristics, modulo arithmetic, is related to the fact that digital computers are traditionally regarded by designers as machines upon which to perform numeric calculations. The second is probably related to the relationship between machine instruction sets and the discipline of circuit design.

The seminal paper in the field of circuit design was written by Claude Shannon as a graduate student at MIT in 1938 [5]. Therein he shows that if E(2) denotes the space {0, 1} and if, for some natural number n, f is a function $f: E^n(2) \rightarrow E(2)$, then f may be represented in the form:

$$f(x_1, x_2, \ldots, x_n) = \Sigma\, a_i x^*_1 x^*_2 \cdots x^*_n \qquad (1)$$

where for all j ($1 \leq j \leq n$) x^*_j is either x_j or $\overline{x_j}$, the generalized summation sign represents disjunction, and the implied multiplication is conjunction.

The joys and sorrows of working with the disjunctive normal representation of a function are well known, as is the fact that a representation in the form (1) is not unique.

The above result follows immediately from the theorem which has come to be called the Shannon Decomposition Theorem, namely, if f is defined as above then there are two functions g: $E^{n-1}(2) \to E(2)$ and h: $E^{n-1}(2) \to E(2)$ such that:

$$f(x_1, \ldots, x_n) = x_n g(x_1, \ldots, x_{n-1}) + \overline{x}_n h(x_1, \ldots, x_{n-1}).$$

These results have dominated the area of circuit design for its whole history.

Another decomposition theorem was published in 1954 by Reed [6] and Muller [7], which has come to be called the Reed-Muller Decomposition Theorem. Reed proves that if f is again defined as above and if $\oplus$ denotes non-equivalence (exclusive-or), then there exist functions g: $E^{n-1}(2) \to E(2)$ and h: $E^{n-1}(2) \to E(2)$, such that:

$$f(x_1, \ldots, x_n) = g(x_1, \ldots, x_{n-1}) \oplus x_n h(x_1, \ldots, x_{n-1}).$$

Seldom noted is the fact that Reed explicitly refers to deriving the formula using classic difference methods, that the method depends upon the fact that E(2) under the operations of non-equivalence and conjunction form a field with two elements. Reed notes that the method may be generalized to any finite field.

There are two striking differences between the Shannon theorem and the Reed-Muller theorem:

1.) Shannon Decomposition uses the logical operations;
Reed-Muller Decomposition uses two field operations;

2.) Shannon Decomposition may be generalized to a space of arbitrarily many elements, but the number of operators required increases linearly with the size of the space,
Reed-Muller Decomposition may be generalized to a space of $k = p^q$ elements, where p is a prime and q is a natural number, and the number of operators required remains two.

It is with a generalization of Reed's and Muller's work to instruction sets that we are concerned in this paper.

II. An Environment

We assume a hypothetical machine M of fixed word size W, not necessarily binary-based but at least p-based where p is a prime. We assume that the instruction set of M includes addition and multiplication over the field $GF(p^W)$, the field of $k = p^W$ elements. Let $E(k) = \{ 0, 1, \ldots, k-1\}$. We assume that the Machine M is to be used to evaluate functions of the form $f: E^n(k) \rightarrow E(k)$ for natural numbers n.

The question which concerns us is: given this machine M, how would you program for it? (A first answer probably should be: slowly and with much pain.) By "program" we mean the process of firmware/hardware interface to provide a user with an instruction set which is useable, familiar, some how pleasant.

The main point of this paper is that if the hardware is as assumed above, the process of firmware/hardware interface can itself be automated.

Something needs to be said, before we proceed, about the reasonableness of the above hypothetical environment. Except for certain experimental machines, digital computers have been binary. Ternary machines have been discussed at length; ternary circuits have been designed, but have always been prohibitively expensive to implement. Recently Mouftah and Jordan at Laval [8] and Etiemble and Israel at Paris VI [9] have built ternary logic devices using off-the-shelf chips. In the light of this it does not now appear that ternary (and even quinary) machines can be as easily dismissed as unrealistic. Further, circuits for Galois addition and multiplication have been designed and implemented for many years [10, 11, 12]. Most unrealistic is our limitation of the problem domain of concern to problems of function evaluation. This stems from the problems of incarnating primitives for, say, string processing and list processing into hardware.

III. Mathematical Results.

Throughout this section and the section which follows we use + and implied multiplication to denote addition and multiplication over a field $GF(p^W)$. We denote addition and multiplication of integers modulo k by x+y(mod k) and x*y(mod k) respectively. Subtraction and division are over $GF(p^W)$.

The first two results concern the representation of a function as a polynomial over a finite field.

If f: E(k) → E(k) is a one-place function, then there is a polynomial in one indeterminate which defines f, specifically:

$$f(x) = \sum_{i=0}^{k-1} a_i x^i. \qquad (2)$$

This representation is unique.

If $g: E^2(k) \to E(k)$ is a two-place function, then there is a polynomial in two indeterminates which defines g, specifically:

$$g(x,y) = \sum_{i,j=o}^{k-1} a_{ij} x^i y^j . \tag{3}$$

This representation is unique.

These results immediately generalize to n-place functions. The uniqueness of the representation is a radical departure from the situation in the case of the representation of a function in disjunctive form. However the uniqueness gained is useless unless there is an effective computational means to calculate the coefficients a_i and a_{ij}.

The computational means is provided by the divided difference methods of Newton [13] . In the usual works on finite difference methods [14, 15] the methods are developed for the real or rational fields, but the modification of the techniques to finite fields is direct.

Firstly, we give the formulae for the representation of one-place functions. Let $x_o, x_1, \ldots, x_{k-1}$ be any permutation of the elements of E(k). Define a difference operator as follows:

$$D_x f(x_j) = \frac{f(x_j) - f(x_{j+1})}{x_j - x_{j+1}} ;$$

$$D_x^p f(x_j) = \frac{D_x^{p-1} f(x_j) - D^{p-1} f(x_{j+1})}{x_j - x_{j+p}} . \tag{4}$$

This difference operator is easily implemented as a recursive procedure.

Newton's Theorem is:

$$f(x) = f(x_o) + \sum_{i=o}^{k-1} D_x^i f(x_o)(x-x_o) \ \ldots \ (x-x_{i-1}). \qquad (5)$$

The polynomial must be expanded to obtain the form of (2) above.

If $f(x,y)$ is a two-place function and if $x_o, x_1, \ldots, x_{k-1}$ and $y_o, y_1, \ldots, y_{k-1}$ are two (possibly distinct) permutations of $E(k)$, then define:

$$D_x^1 f(x_j,y_{j*}) = \frac{f(x_j,y_{j*}) - f(x_{j+1},y_{j*})}{x_j - x_{j+1}} \qquad (6)$$

$$D_x^p f(x_j,y_{j*}) = \frac{D_x^{p-1} f(x_j,y_{j*}) - D_x^{p-1} f(x_{j+1},y_{j*})}{x_j - x_{j+p}}$$

$$D_y^1 f(x_j,y_{j*}) = \frac{f(x_j,y_{j*}) - f(x_j,y_{j*+1})}{y_{j*} - y_{j*+1}}$$

$$D_y^p f(x_j,y_{j*}) = \frac{D_y^{p-1} f(x_j,y_{j*}) - D^{p-1} f(x_j,y_{j*+1})}{y_{j*} - y_{j*+p}}$$

In this case, Newton's Theorem is:

$$f(x,y) = f(x_o,y_o) + \sum_{i=o}^{k-1} D_x^i f(x_o,y_o)(x-x_o) \ \ldots \ (x-x_{i-1}) \qquad (7)$$

$$+ \sum_{i=o}^{k-1} D_y^i f(x_o,y_o)(y-y_o)\ldots(y-y_{i-1})$$

$$+ \sum_{i,j=o}^{k-1} D_x^i D_y^j f(x_o,y_o)(x-x_o)\ldots(x-x_{i-1})(y-y_o)\ldots(y-y_{j-1}).$$

Provided that the recursive procedures for evaluating D_x and D_y are provided with a remembrance of the values already calculated, the method above is simple and computationally effective. In the forms (5) and (7) the representation of the function depends upon the particular permutation of the elements x_i and y_j chosen. When the polynomials are multiplied out to obtain the forms (2) and (3), the resulting polynomial is unique.

IV. Implementation.

A Galois field $GF(p^q)$ is a vector space of dimension q over the field GF(p), which coincides exactly with the ring of integers modulo p. Addition of the elements of $GF(p^q)$ is component-wise. In order to define multiplication it is necessary to choose a polynomial P(x) of degree q which is irreducible over GF(p). Two elements of $GF(p^q)$ are multiplied as if the coordinates were coefficients of polynomials, the result being reduced modulo P(x).

In general there is a wide variety of choice for the polynomial P(x). The professional algebraist assures us that all of the fields resulting from different choices of P(x) are isomorphic. Isomorphism appears to be totally unrelated to simplicity of implementation. Different choices of the polynomial P(x) can have a great effect on the complexity of the implementing circuitry. Complete tables of polynomials irreducible over GF(2) for orders to 16 are in [16]. Complete tables of polynomials irreducible over GF(3) for orders to 10 are in [17]. The most complete tables for GF(5) and GF(7) (to orders 5 and 4, respectively) are in [18].

The table below contains some information about the results of the application of the methods outlined herein to two situations. The polynomials corresponding to six functions were evaluated for the field GF(16) and for the field GF(9). The table below presents the results. The column "UN" refers to the unnormalized form of the polynomial generated ((5) or (7) above); the column "N" refers to the normalized polynomial ((2) or (3) above). The integer shown is the number of non-zero coefficients in the corresponding polynomial. The last two functions are defined respectively as:

$$\text{Signum}(x) = \begin{cases} 1, \text{ if } 0 < x \leq (k-1)/2; \\ 0, \text{ if } x = 0; \\ k-1, \text{ if } (k-1)/2 < x \leq k-1. \end{cases}$$

(That is, the usual association of the high integers with the negative integers.)

$$\text{Order } (x,y) = \begin{cases} 1, \text{ if } x < y; \\ 0, \text{ if } x = y; \\ -1, \text{ if } y < x. \end{cases}$$

(Here the high integers are again taken to represent negative integers and, in particular, k-1 to represent -1.)

	GF(16)		GF(9)	
	NP	N	NP	N
x+y(mod k)	124	124	17	18
x*y(mod k)	129	174	17	21
x+y(mod k-1)	134	233	42	69
x*y(mod k-1)	161	206	50	48
Signum(x)	15	5	8	4
Order(x,y)	184	163	57	55

Specifically, in GF(16) we have:

$$\text{Signum}(x) = 14x + 14x^2 + 14x^4 + 14x^8 + x^{15},$$

and in GF(9) we have:

$$\text{Signum}(x) = 5x^2 + 5x^4 + 5x^6 + x^8.$$

These values are, at best, depressing for the future of a direct simple implementation of the method to words of 8 or 16 bits.

V. A Future Direction.

In the above work we have limited ourselves by using a polynomial representation of functions. Since in a field of characteristic 2 addition and subtraction coincide and since in a field of characteristic 3, $x+x = -x$, there would be no point in providing subtraction as a fundamental operation for our hypothetical machine. It may be profitable to add division to our set of primitive operations.

This may be more precisely formulated in the following way. Given $f(x,y)$, how can one find polynomials $P(x,y)$ and $Q(x,y)$ such that the following conditions all hold:

1. the degree of $P(x,y)$ is strictly less than the degree of $Q(x)$;
2. the degree of $Q(x,y)$ is strictly less than k;
3. $Q(x,y)$ is either irreducible or a product of irreducible factors (and hence never 0);
4. $f(x,y) = P(x,y)/Q(x,y)$.

It may be that this modification will render the Galois field primitives a viable instruction set for incorporation into hardware.

In addition to this, the work reported at Micro-7 by Louise Jones [19] concerning sets of control primitives needs to be extended so that suitable control primitives can be linked to the state modifying primitives discussed in this paper.

Table I -- Operations for HALL

Mnemonic	Operation	Code (hex)
Arithmetic Unit		
NO	No operation	0
+B	Binary addition X + Y	1
-B	Binary subtraction X - Y	2
+D	Decimal addition X + Y	3
-D	Decimal subtraction X - Y	4
AN	And X Y	5
OR	Or X Y	6
EX	Exclusive or X Y	7
SL	Shift X left one bit	8
SR	Shift X right one bit	9
Status Unit		
BITO	Set bit to 0	0
BITI	Set bit to 1	1
IBIT	Invert bit	2
DIGO	Set digit to 0	3
DIGI	Set digit to 1	4
IDIG	Invert digit	5
NOOP	No action	7
BZHO	Set bit Z = 0*	8
BZLO	Set bit A = 0	9
DZIO	Set digit Z = 0	A
IBZO	Invert bit Z = 0	B
BZHD	Set bit $0 \leq Z \leq 9$**	C
BZLD	Set bit $0 \leq A \leq 9$	D
DZID	Set digit $0 \leq Z \leq 9$	E

---

* Set bit to 1 if Z = 0.

** Set bit to 1 if Z is a digit, i.e., is a number between 0 and 9.

[2, p. 224]

Table II -- The Instruction Set Proposed for the EDVAC

Instructions consist of <arithmetic instruction> <variation> .

They operated on registers I, J, and A.

Arithmetic Instructions

AD	Set A ← I + J.
SB	Set A ← I - J.
ML	Set A ← A + I x J (rounded)
DV	Set A ← I/J (rounded)
SQ	Set A ← $\sqrt{I}$ (rounded)
II	Set A ← I
JJ	Set A ← J
SL	If A $\geq$ 0, set A ← I; if A < 0, set A ← J.
DB	Set A ← binary equivalent of decimal number I.
BD	Set A ← decimal equivalent of binary number I.

Variations

H	Do the operation as described above, holding the result in A.
A	Do the operation as described above, then set J ← I, I ← A, A ← 0.
S	Do the operation as described above, then store the result A into memory location yx and set A ← 0.
F	Do the operation as described above, then store the result into the word immediately following this instruction, set A ← 0, and perform the altered instruction.
N	Do the operation as described above, then store the result into the work immediately following this instruction, set A ← 0, and skip the altered instruction. [4, pp. 250-1]

Bibliography

1. Joe Weisbecker, "A Simplified Microcomputer Architecture", IEEETC (March, 1974) pp. 41-7.

2. R. H. Evans, L. H. Moffett, R. E. Merwin, "Design of Assembly Level Language for Horizontal Encoded Microprogrammed Control Unit", Micro-7 Preprints (September, 1974) pp. 217-224.

3. A. J. Nichols, III, "A Microprogramming Framework for Experimental Machine Design", SIGMICRO Newsletter, (July, 1971) pp. 17-21.

4. Donald E. Knuth, "Von Neumann's First Computer Program", Computing Surveys, (December, 1970), pp. 247-60.

5. Claude E. Shannon. "A Symbolic Analysis of Relay and Switching Circuits", Trans. Am. Inst. Elec. Eng. 57 (1938), pp. 713-23.

6. Irving S. Reed, "A Class of Multiple-error-correcting codes and the Decoding Scheme", Trans. IRE - Info. Theory PGIT-4 (Sept. 1954), pp. 38-49.

7. D. E. Muller, "Application of Boolean Algebra to Switching Circuit Design and to Error Correction", IRE Trans. - Elec. Comp. EC-3, No. 3 (Sept. 1954), pp. 6-12.

8. H. T. Mouftah and I. B. Jordan, "A Design Technique for an Integrable Ternary Arithmetic Unit", Proc. 1975 Int. Symp. on Multiple-valued Logic, (Bloomington, Indiana, May 1975), pp. 359-372.

9. D. Etiemble and M. Israel, "Implementation of a Complete Ternary Algebra - Application to Ternary Flip Flop", Proc. 1975 Int. Symp. on Multiple-valued Logic, (Bloomington, Indiana, May 1975), pp. 316-329.

10. James T. Ellison, Universal Function Theory and Galois Logic Studies (ARCRL-72-0109) (Bedford, Mass.: Air Force Cambridge Research Laboratories, 1972).

11. B. A. Christensen, J. T. Ellison, R. A. Eggan, Galois Polynomial Generation (PX-7703) (St. Paul: Sperry Rand-Univac, 1972).

12. B. A. Christensen, Notes on Galois Logic Design (PX-10452) (St. Paul: Sperry Rand-Univac, 1973).

13. Isaac Newton, "Approaches to a General Theory of Finite Differences" [1675-6] in D. T. Whiteside (editor), The Mathematical Papers of Isaac Newton, vol. 4, (Cambridge, The University Press, 1971), pp. 14-73.

14. Charles Jordan, Calculus of Finite Differences, (New York, Chelsea Publishing Company, 1947).

15. L. M. Milne-Thomson, The Calculus of Finite Differences, (London, Macmillan and Co., 1933).

16. W. W. Peterson, Error-Correction Codes, (New York, John Wiley and Sons, 1961), pp. 472-492.

17. David P. Wolff, Irreducible Polynomials over GF(3), unpublished M.S. probject, Virginia Polytechnic Institute and State University, (June 1975) pp. 25-57.

18. Randolph Church, "Tables of Irreducible Polynomials for the First Four Prime Moduli", Annals of Math. 36, no. 1 (January 1935), pp. 198-209.

19. Louise H. Jones, "Microinstruction Sequencing for Structured Programming", Micro-7 Preprints (September, 1974) pp. 277-89.