Technical Report CS74020-R

THE LOGICAL FOUNDATIONS OF MICROLANGUAGES

Thomas C. Wesselkamper

November 1974

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

Abstract

After the consideration of two recent examples of instruction sets for microprogrammable computers, the article sketches known and new results about complete sets of functions which appear to be applicable to micro-language development. Some needed areas of research are pointed out. Functional completeness is linked to research in control primitives for machines.

I. The Problem

A. Electronic digital computers began in a world of vacuum tubes and diodes. They were endowed with sets of machine instructions heavily influenced by the available electronics of the time. It is not possible that anything else could have occured.

In nearly three decades since, technology has changed mightily. Machines have become word addressable; their electronic components moved through transistor technology into integrated circuit technology. Sets of machine instructions have grown: IBM's System/370 features over one hundred fifty instructions.

Recent years have witnessed a growing interest in microprogramming. Often this represents a return to the consciousness that there is a machine beneath all those layers of high level language.

This interest in microprogramming and the correlative ability to easily emulate machines in what has come to be called firmware makes it possible to ask a set of questions which (logically) should have been asked before the first machine was constructed. It is now possible to ask: What should a machine be able to do?

B. Two recent examples exhibit the current state of the pragmatic answer to this question. The first 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 add, 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. 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. [11] 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.

II. General Purpose Computers and Complete Sets of Functions

A. This paper examines some of the possible answers to the question: What should a machine be able to do? We are concerned that while one hundred fifty instructions are quite likely to be sufficient for any job, they are not likely to be necessary. Even if one argues that they are desirable at some level of machine definition, they are not desirable at the microlevel.

We assume that when a computer manufacturer says that his machine is a "general purpose machine" he means that it is functionally complete over the space of its words.

More specifically, we assume a hypothetical machine is fixed word size, not necessarily binary. The value of a word ranges over a set $E(k) = \{0, 1, \ldots, k-1\}$. A machine M with words in E(k) is complete if whenever n is a natural number and $f: E^n(k) \rightarrow E(k)$ is a function over E(k) then/for any set of values $(x_1, x_2, \ldots, x_n)$ it is possible to evaluate $f(x_1, x_2, \ldots, x_n)$ on M.

This notion of completeness is implied by, but not necessarily equivalent to, the completeness with constants of the set A of those machine operations which have the property of being functions over E(k). This is opposed to those machine operators which are control operators, for example, a branch instruction. In Section III of this paper we survey some of the known results about complete sets of functions.

B. In the practical area of machine design two other questions arise in the process of evaluating a proposed instruction set:

(1) Can normal human beings (as opposed to multiple-valued logicians) write programs using the proposed instruction set? The pragmatists of the last thirty years must be granted that their instruction sets (however inelegant) have been useful to write programs. Human factors suggest that the functions chosen need to have intuitively simple definitions and need to possess "nice" algebraic properties, such as associativity and commutativity.

(2) What are the minimal storage and time requirements needed to evaluate on machine M a given set of functions over E(k)? From the point of view of machine design the optimal set of functions is not a minimal set, but a set that minimizes time and storage requirements over some subset of functions which are "interesting" in one or more applications areas.

III. Some Theoretical and Experimental Results

A. This author investigated an instruction set with a single operator:

$$Sxyz = \begin{cases} z, & \text{if } x = y; \\ x, & \text{if } x \neq y. \end{cases}$$

For any natural number k, this function is complete over E(k). [4] The choice of operator was inspired by the work of Markov. It is possible (if nerve-racking) to program this way. The writer refrained from publishing the fact that for some one place functions over E(k), the program to evaluate the function takes up at least 19k words of storage.

B. At this Symposium in 1974, this writer showed that for a natural number k, there exists a set of three abelian semigroup operations on E(k) such that the set of functions defined by these operations is complete. [5] Dr. J.C. Muzio has since reduced the number to two and this writer has shown that it cannot be reduced to one.

Specifically, this author showed that if:

$Axy = x + y \pmod{k}$;

$Mxy = xy \pmod{k}$; and

$$Jxy = \begin{cases} 0, & \text{if } x = 0 \text{ or } y = 0, \text{ not both}; \\ 1, & \text{otherwise}; \end{cases}$$

then {A, M, J} is complete with constants over E(k). Muzio has shown that {A, J} is complete with constants.

An attempt was made over the last year to program with the set {A, M, J}. All went well until it was necessary to use some property related to the ordering of E(K) obtained by relativizing the usual ordering of the integers to E(k). Most

often the user wants, not $E(k) = \{0, 1, \ldots, k-1\}$, but rather $E'(k) = \{-[k/2], \ldots, 0, \ldots, [(k-1)/2]\}$, (where square brackets denote the "greatest integer" function).

C. Order can be handled nicely by introducing the "signum" function:

$$S*x = \begin{cases} -1, & \text{if } -[k/2] \leq x \leq -1; \\ 0, & \text{if } x = 0; \\ 1, & \text{if } 1 \leq x \leq [(k-1)/2]. \end{cases}$$

In the usual infix notation we have:

$$Jxy = (S*((S*xS*y)^2 + (S*x)^2 + (S*y)^2 - 1))^2;$$

for if $x = y = 0$, then $Jxy = (S*(0 + 0 + 0 - 1))^2 = 1$;

and if $x = 0$, $y \neq 0$, then $Jxy = (S*(0 + 0 + 1 - 1))^2 = 0$;

and if $x \neq 0$, $y \neq 0$, then $Jxy = (S*(1 + 1 + 1 - 1))^2 = 1$.

Hence the set $\{A, M, S*\}$ is complete with constants.

The results of programming with $\{A, M, J, S*\}$ have been very pleasant. This appears to be closer to the notion of an optimal set than any of the minimal sets tried. It has not appeared practical to attempt to use only $\{A, M, S*\}$.

D. If $k = p^n$, for some prime p and natural number n, then the set of functions $\{f, g\}$ defined by:

$$fxy = xy; \quad gxy = x + y;$$

where addition and multiplication are defined over $GF(p^n)$, is complete with constants. In fact the polynomials in j indeterminates with exponents in the range [0, k-1] uniquely represent the functions: $E^j(k) \to E(k)$.

There has been extensive work in the field of logic design using Galois operations as primitives [6, 7, 8].

As was the case above with the ring Z(k), the problem of inducing an order relation onto $GF(p^n)$ is formidible, that is, the polynomial which corresponds to the order relation:

$$x < y$$

is very long. It appears that the addition of an ordering function might be on the path to optimizing the set of functions.

Since the great body of classical work on divided difference methods is applicable in any field, work with Galois operations has a place to start.

All present computers are either binary or ternary. Hence the Galois fields involved are of characteristic 2 or 3, respectively. This suggests that nothing is to be gained by using subtraction as a primitive. The same is not clear about the inclusion of division as a primitive operation. Programming with rational functions might be far easier than with polynomials alone. There appears to be little research in this area. This investigation into rational functions appears to this writer to be an important research direction.

E. It appears to be only of theoretical interest that every simple nonabelian group is complete. [9]

IV. From Functions Through Algorithms to Programs

It is not realistic to develop a computer which evaluates functional expressions of arbitrary length. The fundamental notion of a digital computer links it to the notion of an algorithm.

An algorithmic language requires, in addition to the logical operations of the kinds treated earlier in this paper, some instructions which control the flow of the program realizing an algorithm. It is sufficient to provide a "goto"

operation which provides transfer of control to a location specified by its argument. This must be accompanied by the labelling of statements. The complete syntax for such a language which the writer has successfully used is contained in Table III.

Flow of control primitives are as important as logical primitives in the design of an optimal language. Their analysis does not appear to be within the scope of classical multiple-valued logic.

The most important study of the effect of the choice of control primitives on the optimality of a language appears to be that of Dr. Louise Jones [10].

V. General and Specific Research Directions

A. Released by IC technology from the old constraints on possible logic and control primitives, how do we develop a set of primitives which are easy to use and which, when applied to known and future applications areas, produce efficient algorithms in terms of size and length of computation?

B. What would be good primitives for list-processing? for string-handling?

C. Specifically, how may the ring $Z(p^n)$ be represented over $GF(p^n)$, both as polynomials and as rational functions? Can ordering be achieved inexpensively as a rational function?

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 $0 < A < 9$	D
DZID	Set digit $\overline{0} \leq \overline{Z} \leq 9$	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 word immediately following this instruction, set A ← 0, and skip the altered instruction. [11, pp. 250-1]

Table III -- The BNF Grammar for a Rather Primitive Language

```
<program>            ::= <decl list> / <statement list>
<decl list>          ::= <decl> | <decl> <decl list>
<decl>               ::= <allocation> | <initialization>
<allocation>         ::= dec <dec identifier> | dec <array designator>
<array designator>   ::= <dec identifier> (<number>)
<dec identifier>     ::= <identifier>
<initialization>     ::= <allocation> <number>
<statement list>     ::= <statement> | <statement> <statement list>
<statement>          ::= <label> : <simple statement>
<label>              ::= <number>
<simple statement>   ::= <assignment> | <goto>
<assignment>         ::= store <arg> at <name>
<goto>               ::= goto <arg>
<arg>                ::= <expression> | <name> | <number>
<expression>         ::= <op> <arg> <arg>
<op>                 ::= A | M | J | S*
<name>               ::= <array element> | <identifier>
<array element>      ::= <identifier> (<number>)
```

Bibliography

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

2. R. H. Evans, L. H. Moffett, and 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. T. C. Wesselkamper, "A Sole Sufficient Operator", NDJFL, (January, 1975) pp. 86-88.

5. T. C. Wesselkamper, "Some Completeness Results for Abelian Semigroups and Groups", Proceedings of the 1974 International Symposium on Multiple-valued Logic, (May, 1974) pp. 393-400.

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

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

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

9. Heinrich Werner, "Finite Simple Nonabelian Groups are Functionally Complete", Notices AMS, (August 1973) (*73T-A228), p. A-561.

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

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