# STORED LOGIC REVISITED

by Richard H. Hill

Information Systems Department
Thompson Ramo Wooldridge Inc.
RW Division

December 4, 1961

Canoga Park, California

A paper delivered to the Los Angeles Chapter of the Association for Computing Machinery, December 6, 1961.

Stored logic is not a new concept. The Stored Logic idea has its roots in the concept of micro-coding, expounded early in "modern" or post-ENIAC computing history. It is my purpose here to describe the RW implementation of these ideas, but since it was both refreshing and instructive to rummage about in computing history for some background, this introductory section presents the gleanings of a hasty research on the micro-coding concept.

M. V. Wilkes in England was apparently the first to devote serious attention to the potential of micro-coding. In a paper delivered at the Manchester University Computer Inaugural Conference in July, 1951, titled "The Best Way to Design an Automatic Calculating Machine", Wilkes argued for the micro-coding approach.(1) Wilkes and Stringer further expanded these ideas in a paper published in England in 1953 under the title "Micro-Programming and the Design of the Control Circuits in an Electronic Digital Computer". (2) About this same time in the United States, "Tommy" Tompkins, now at UCLA but then with ERA, espoused microprogramming for the design of a special purpose machine to handle permutations. (3)

The wellspring of this thinking appears to be a variant on the theme proposed in Burks, Goldstine and von Neumann's famous memorandum of 1946 that started this whole bloody mess we call stored programming.(4) In this memorandum the advantage of instructing a computing machine with numbers was held to be the ability of the machine to modify its own program in the course of execution. As computers have developed we have learned to appreciate this advantage, but principally in terms of the ability to modify operand addresses. There is considerable internal evidence to the effect that Burks, et. al., envisioned freely modifying the order code as well. Micro-coding is a logical extension of this principle.

Order codes quickly grew beyond such amenable proportions, however. As programs were made to accommodate more, and more diversified, instructions the need to modify order codes seemed to diminish.

Wilkes' ideas, much too complex to examine here, envisioned coding on the micro-micro level, if there is such a thing. Bits of an order code would be assigned to the gates of a decoding matrix, much as is done in a conventional computer, but the matrix itself would be variable. Thus not only would the operations be decoded according to the effect of the order code on the gates of the decode matrix, but the program could alter the state of the matrix so that a given order code interpreted at one point in time would cause completely different actions when interpreted at another point in time.

Skipping along through the years, we encounter "A Note on Microprogramming" in the April, 1956 issue of the Journal of the ACM, by Herbert T. Glantz. (5) The article might better be titled "A Manifesto for Microprogramming". Glantz observes that the control logic of a computer typically combines sets of subcommands to produce the "machine language" instructions. One computer, he notes, uses subsets of a set of 35 subcommands to produce 62 machine instructions. Programmer access directly to the subcommands would eliminate the need to overgeneralize machine instructions and would give the programmer unlimited flexibility in the preparation of special analytic instructions. He states: "Each special problem could conceivably have its own special purpose computer which would have an order code hand tailored for efficient solution of this particular case. For some jobs one or two special instructions might serve to greatly reduce the running time for machine solution. Further, a 'general purpose computer' could also be easily made available for the casual user or for the problem not requiring special care and planning." One solution to the problem of providing special order codes is through compilers, Glantz recognizes, and I assume by compiler he means also interpretive routines. Compiler routines have an obvious advantage in programming ease. "However, once the instructions have been compiled into a machine order code, we are faced with two time-consuming defects. One is the repeated memory accesses which are necessary as the machine obeys the compiled list of instructions. This defect is compounded by the necessarily less-than-optimum memory assignments which is effected by the executive compiling routine. (Note that he is discussing drum machines.) The second effect is the strong possibility of the repeated performance of superflous subcommands which serve no useful function in the particular computation involved.

"Both of these defects are obviated by the use of microprogramming techniques. On one hand only one memory access is called for and thereafter all performance is carried out in the hardware, and in the second case only the required subcommands are carried out. Furthermore, the substitution of one microprogrammed instruction for many machine instructions will permit a reduced usage of slower secondary memory with a corresponding increase in performance speed."

I hope these quotations do not do Mr. Glantz too great an injustice. In fairness, although some of the concepts he presents seem anachronistic in this era of large fast-access memories and sophisticated control logic, he does recognize both the programming difficulties inherent in the micro-coding approach and the need to provide pre-programmed packages for the poor wight he calls the "casual user". His proposed implementation, however, does not fully carry out the general approach to the theme. The machine he outlines is a conventional computer with a micro-programmed attachment, as it were; a special set of hardware that could be used optionally for microprogramming.

So far as I am aware, one of the very early practical approaches to microprogrammed hardware had its beginnings about the same time that Glantz's article appeared. The April, 1957 issue of the ACM Journal carries a report by Robert J. Mercer, (then with Thompson Ramo Wooldridge, RW Division) that proposes a feasible machine.(6) Mercer built heavily upon the work of Wilkes and Tompkins, in particular, adopting the latter's philosophy but the hardware notions expressed by Wilkes. He was aware of Glantz's opinions, but appears to have given them little more than a passing Glantz, so to say. In fact, Mercer began to develop his ideas at UCLA prior to joining RW.

A basic premise of the micro-programmed machine, according to Wilkes and Tompkins, is the wholeness or integrity of operations occuring within a single clock pulse. In this concept a number of elementary operations may occur simultaneously, but independently; and especially, the operations occuring on a given clock pulse are not dependent upon the results of a previous pulse (except for certain control functions, such as branch or compare) nor do they establish conditions for following sequences. Tompkins had written that a micro-operation is any operation "whose total course of action is dictated by pulses occuring simultaneously without requirement for later generation of stimulus pulses."

Now, let's see if we can simplify and expand these remarks. Each clock pulse is a self-contained event. Clock pulses must be long enough, then, to allow reasonably significant events to occur. In particular, a single clock must be long enough to permit carries to propagate through a full word parallel add. Other events might include register transfers, address incrementations, shift steps, multiply and divide steps, and memory transfers. With the long clock pulse, and with each pulse self-contained, we have inevitably a completely synchronous machine. The AN/UYK-1 (TRW-130) implementation of this synchronizes all micro-operation execution to the memory cycle. A clock pulse is one-half cycle in duration, and a memory read or write is executed on every clock simultaneously with certain other standard micro-operations and whatever optional micro-operations that are called for in the order code. Every internal instruction requires either two or three memory cycles (four or six clock times) for execution, except for iterative commands such as shift and multiply. In these cases the micro-operations of a particular clock repeat a number of times specified in the instruction.

What does this buy us? The primary advantage is a substantial lessening in the amount of control logic required. There are no conditional sequences of control, since the events on each clock pulse are self-contained and predetermined. Thus circuits to establish, hold and break delays are gone, interlocks are virtually nonexistent, and many triggers usually required to hold test conditions may be eliminated.

So there is a very real saving in computer hardware. This saving in the AN/UYK-1 (TRW-130) has approached the optimum, rather than the ideal; Mercer, and Tompkins too, recognized that some compromises with true microprogramming were necessary to produce acceptable operating speeds, as well as to build in the machine control functions that are messy but needed. It's hard to microprogram a command like "get that cotton-pickin' card reader moving," for example.

Not long after Mercer's article appeared, the computing world went way off on the opposite track. If Glantz prepared a manifesto for microprogramming, then certainly Saul Gorn's Letter to the Editor in the inaugural issue of the Communications of the ACM has to be the manifesto of the compilationists. (7) Gorn wrote, in effect, "machine language programmers, break your bonds." Of single address machines he wrote: "Sentences with only one verb and one noun simply do not express big enough thoughts for most people. We want a language system in which each sentence has one or more verbs and an indefinite number of nouns . . . present codes should be considered to be micro-codes ..."

Gorn wanted to achieve efficiency through a flexible order code, and he felt the way to do this was through using the computer as an interpreter before the fact. Of course, he was correct for a large class of applications and users. The four years since Gorn's letter will certainly be known as the Age of the Compiler -- Fortran, Autocoder, Bacaic, IT, Flowmatic, B-Zero, Cobol, and of course Algol in all of its bedazzling variety. Most connoisseurs agree that '60 is the vintage year, I believe.

Now there seems to be, not a swing away from compilers, useful and capacious as they are, but an undercurrent of realization that there may also be something else. Is it significant that the esteemed gentleman who received the presentation award at the last ACM National Meeting spoke on the topic, "There is Still a Place for Interpreters" ?

The RW approach to Stored Logic asserts that in fact there is still a place for interpreters. Program interpretation before execution - the compiler method - and interpretation during execution can achieve a peaceful coexistence, as can machines that have more and more complex order codes and those that rely on an approach lineally descended from the microprogramming proposals we have examined earlier. The objective of RW's efforts is to marry the virtues of these anti-stream concepts and at the same time minimize what inherent defects as might be present. As Machol pointed out last September, interpretive operation has the virtues of simplicity and flexibility in the language sense; additionally it has the virtue of feasibility on machines of any scale.

Why associate stored logic with interpretative operation ? It is certainly not necessary to operate in the interpretive mode in a microporgrammed computer; the user can, if he wishes, write every possible procedure in the basic instruction language of the machine. But as Glantz noted, the "Casual User" wants simplicity as well as flexibility, and interpretive operation, provided the speed penalties are not too great, is the way to give it to him. As we will see, the AN/UYK-1 (TRW-130) is equipped with some special capabilities that make interpretive operation extremely efficient.

So at last we are ready to take a look at the AN/UYK-1 (TRW-130) itself. Physically it is 60 inches high, about 20 inches wide and 16 inches deep. It is housed in an aluminum casting that weights about 500 pounds, of course without the peripheral gear, and is built completely to military specifications. All circuits are solid-state. In the photograph it is shown with the front cover off. The operator's panel appears at the top; immediately below that, normally covered, is the maintenance console. All of the logic cards in the computer are visible in this picture. The back panel wiring is just behind the visible rank of cards, and the power supplies occupy the space in the back half of the machine. The core memory sits on a small shelf just behind the operator's panel.

The memory contains 8,192 fifteen-bit words and can be expanded to 32K. It reads and writes in parallel with an access time of 3 microseconds or a cycle time of 6 microseconds. A clock pulse is also 3 microseconds, thus the clock rate is 1/3 megacycle.

A minimal set of input-output equipment would consist of a typewriter, a paper tape reader and a paper tape punch. The typewriter operates in both directions. These peripheral devices are connected to the computer through a 15-bit parallel information channel. Two other information channels, both 30-bits wide, are provided for communication with devices in the Navy Tactical Data System family, and if not needed for this purpose they can be used with a wide variety of special equipments. A controller unit is available that will allow the AN/UYK-1 (TRW-130) to be used not only with paper tape and typewriter, but also with magnetic tapes, punched card equipment, printers and Teletype send-receive units. Some specialized equipments, including buffers and code translators, have been developed for use with the AN/UYK-1 (TRW-130) on various projects, some of them classified.

Figure 1 shows the logical organization of the machine. The registers -- the illustration shows <u>all</u> of the internal registers except the decode matrix -- have been given deliberately non-committal names because few of them have distinctive functions.

The <u>M register</u> controls all memory accesses except those made through the L register to scratchpad address 0-63. Right here it would be well to point out that most AN/UYK (TRW-130) instructions are addressless. Instead they use an addressing option to refer to the preset contents of the M, A or P registers as an address. Before A or P is used as an address, its contents are exchanged with M, and after use M is always restored. In this way M is used to access operands, and also serves as the Instruction Counter. For this

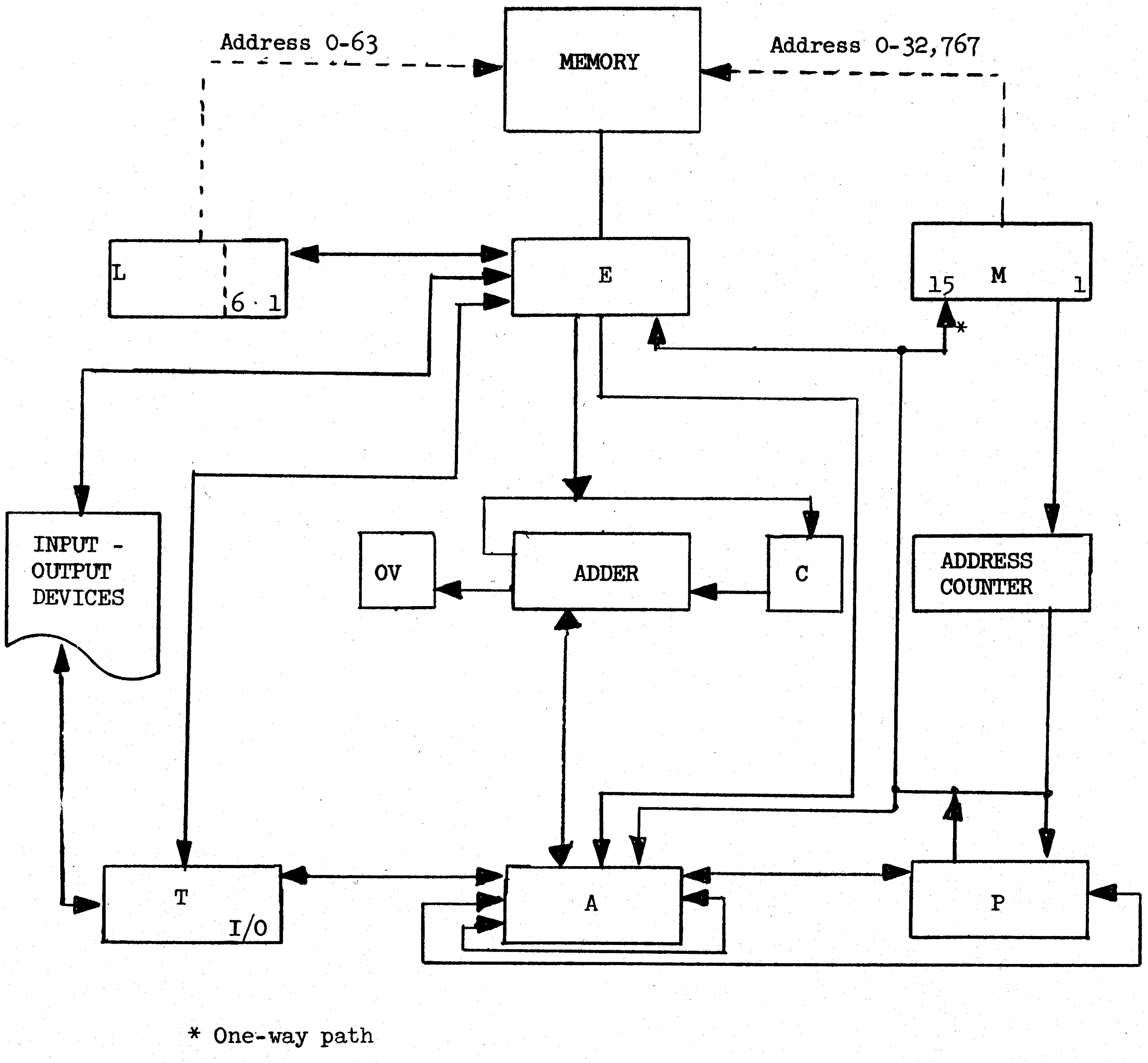


Figure 1 -- AN/UYK-1 (TRW-130) Logical Organization

latter purpose M is always incremented at least once during each instruction.

The E register is the memory exchange register, and is also an input-output register. It holds operands during iterative instructions, and does a few other odd jobs from time to time.

The L register holds instructions during execution, holds operands temporarily between instructions on occasion, and may be used to address the 64 words of scratchpad memory.

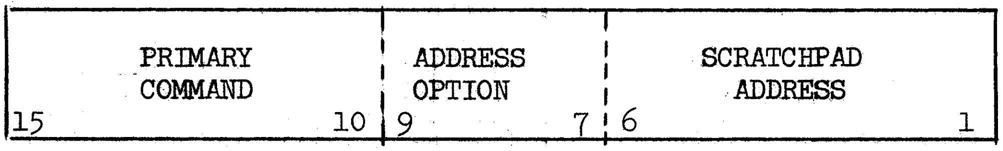
A is the principal arithmetic register, behaving like a conventional accumulator under most circumstances. The contents of A may be used as an address, also.

P is a secondary arithmetic register, usually behaving like a multiplier-quotient register. Its contents may also be used as an address, and frequently P is employed to control a program sequence in the interpretive mode.

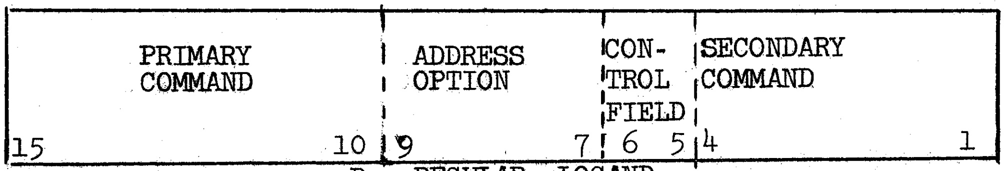
The T register is chiefly used as an input-output buffer for slow-speed devices, but when not needed for this purpose it can be used to store intermediate operands from A or E.

Note that the other registers are a full parallel adder, an overflow indicator and a carry flip-flop. Note also that all transfers from the M register pass through the address counter, where the previous contents of M may be incremented by unity. The programmer controls this function except when the instruction address is incremented.

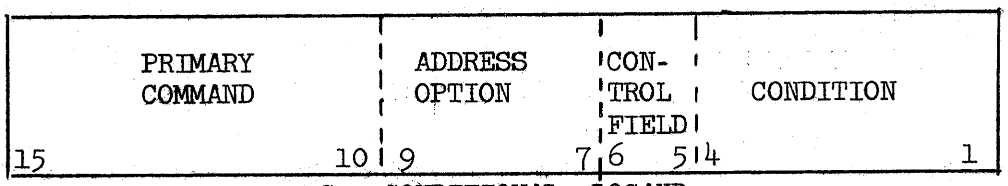
Now it is time to look at the instruction format (Figure 2). The unconventional structure of the machine language instruction seems to invite a new name. Consequently, we have called the machine language commands "logands", a contraction of "logic commands".



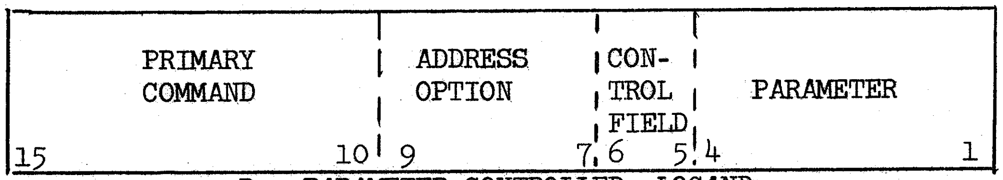
A. DL/IL ADDRESS OPTIONS



B. REGULAR LOGAND



C. CONDITIONAL LOGAND



D. PARAMETER-CONTROLLED LOGAND

(e.g. Shift)

Figure 2 - Logand Formats

The most common logand format is shown first. It contains a six-bit field for the primary command, a 3-bit field for addressing option, a control field, and 4-bits for the secondary command.

There are 32 regular primary commands, 16 of which may also be used as secondary commands. In this format there are six addressing options, direct or indirect from A, P or M. The control option determines whether or not the address used should be incremented by unity, and whether or not a memory access is made. Thus there are 32 x 16 x 6 x 4, or 12,288 different instructions in this format. Some of these are not useful, some are downright dangerous, but most are perfectly acceptable.

The regular commands control register transfers and do useful things like <u>add</u> and the logical functions.

The regular command list is also available for use with the DL and IL address options, in which the secondary command field becomes an address in the scratchpad area of memory. Note that an L address is not incremented. So here we have available 32 x 2 equals 64 single address instructions to add to our list of 12,288 other commands, for a total thusfar of 12,352 logands.

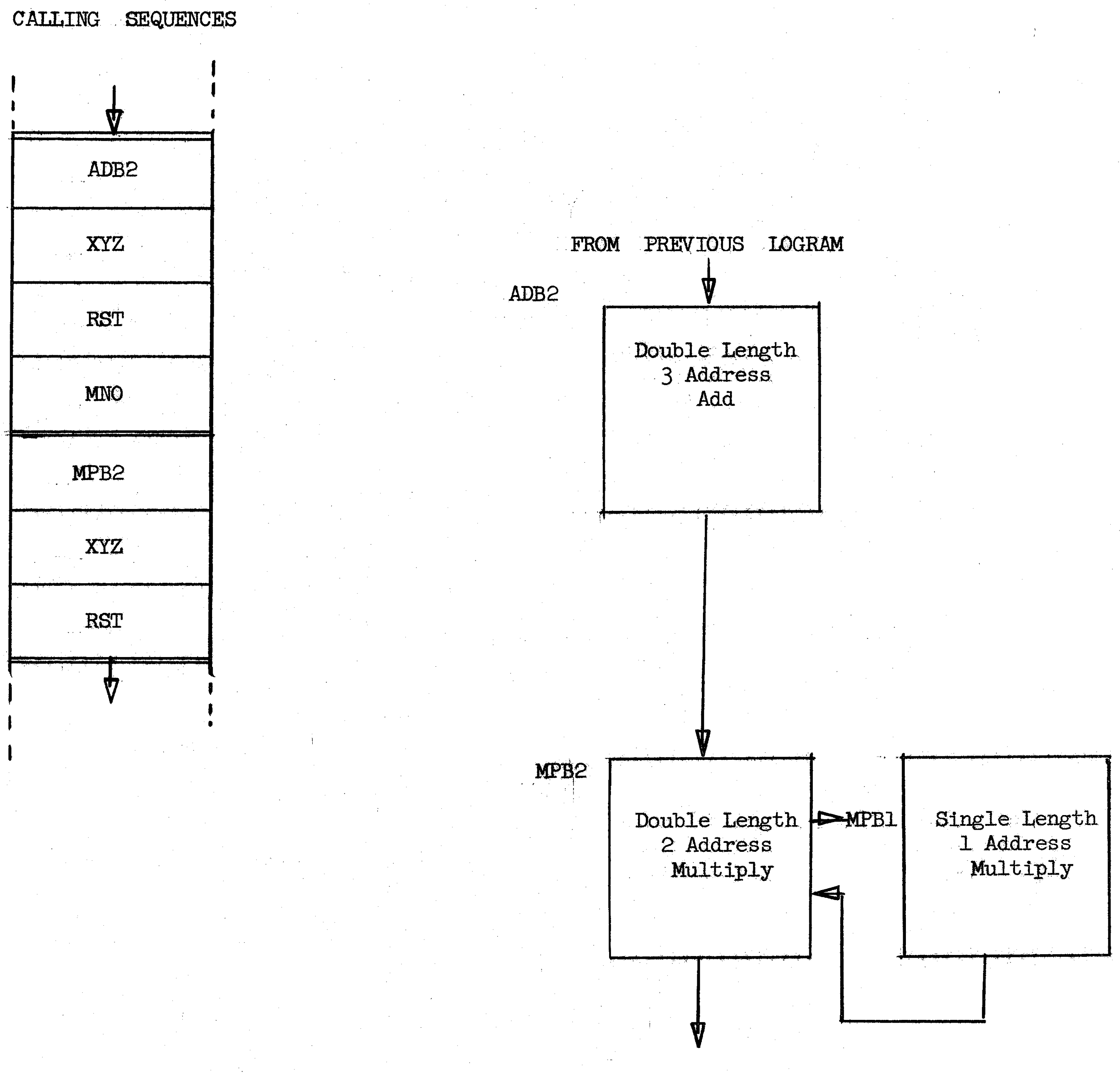
The special logands include the conditional control functions, shifting, multiplication, division, I/O, and so forth, including table search and match functions. There are 22 of these that may be used with the six address options, or 132 more logands, not counting all possible conditions. So we are up to about 132 plus 12,352 equals 12,484 different machine language instructions, without considering all of the conditional possibilities.

Fortunately, of course, the logand structure is such that the logander need not recall each as an entity. He uses fairly simple rules to construct the logand he requires at a given instant. Inherent in the tremendous variety available, though, is unparalleled flexibility. In most instances the logander has available on each cycle exactly the micro-operation needed. This was precisely the objective Wilkes wanted to achieve, and is a major point of Glantz's manifesto. Writing in the language of the machine directly, it is possible to tailor the machine operation quite closely to the problem.

Mostly, though, it is more convenient to program in the interpretive mode, using "instructions" created from sequences of logands. (Figure 3). To distinguish these logands sequences from operational programs, we call them "lograms", a contraction of logic program. Programs, then, are made up of sequences of entries to lograms. Given a logram library, the operational program coder concerns himself only with the preparation of calling sequences.

There is no restriction whatever on what a logram might contain. The AN/UYK-1 (TRW-130) logram library at RW contains perhaps a couple of hundred different lograms. Most of these simulate conventional single address machine functions in both single and double precision data formats. Some are much more extensive. For example: 2, 3, or n address instructions might be used. The library includes a large number of transcendental functions, root-finding and exponential "instructions", coordinate transformations and data conversion operations. For another stored logic machine one of our programmers has written a generalized four-tape merge-sort logram. As the logram library grows, each programmer, of course, has a richer "instruction set" to draw upon.

Interpretive operation typically is a function rather extravagant in its use of machine time. The interpretive executive routine must keep track of the interpretive program sequence, and frequently must go through much machination to deliver to the proper interpretive subroutine the operands called



CALLING SEQUENCES
ADB2
XYZ
RST
MNO
MPB2
XYZ
RST
FROM PREVIOUS LOGRAM
ADB2
Double Length 3 Address Add
MPB2
Double Length 2 Address Multiply
MPB1
Single Length 1 Address Multiply

Figure 3 -- Logram Calling Sequences

out in the program. In the RW stored logic machines, on the other hand, there is usually no time lost in finding operands, and the logram sequence can be preserved automatically without loss of time. In fact there is no interpretive routine under present programming schemes, nor does there appear to be a need for one. We have taken one precaution, however, in the scratchpad memory a cell is set aside by convention to serve as a pseudo instruction counter. It is updated by every logram, principally to provide a trail in case of difficulty with the program. The "overhead" for this averages 12 microseconds per logram, regardless of the logram's length or complexity.

In our experience with stored logic machines, we have found it convenient to simulate one, two, or three address machines, variable-length accumulators, indexing and indirect addressing - these latter come virtually free - and to use almost any data format involving an integral number of words. Single and double precision binary fixed point operations have been fully logrammed. Floating point operations are the next extension. BCD operations have been logrammed in wide variety for another stored logic machine. And, of course, there is a profusion of the functional operations mentioned earlier. Plans for software entail further extensions to the logram library, an assembly routine, a full set of utility routines, and diagnostic programs for maintenance use. A considerable amount in each of these areas already exists - for example, there is an assembly and computer simulation routine for the AN/UYK-1 (TRW-130) now operational on the 7090.

The AN/UYK (TRW-130) is fitted with other capabilities, not directly related to stored logic, that make it of interest. For example, to facilitate multiple precision arithmetic there are four separate add commands (this is a complement machine, thus no subtract). These commands differ only in the way the carry and overflow triggers are treated, and are designed to handle the low-order, intermediate or high order multiple precision operand parts. There is also an add single, for single precision work.

The AN/UYK (TRW-130) also has an extensive interrupt facility directed toward input and output operations. This capability, coupled with reliability and reasonable cost, has led directly to use of the AN/UYK (TRW-130) in a variety of on-line situations. One example of this use is in the Atlantic Missile Range in real-time missile tracking operations.

But the real concern here is with stored logic. How well is it proving out in practice ?

The basic single precision add time in the AN/UYK (TRW-130) is twelve microseconds, or one direct logand cycle. In this length of time an operand is read from the cell addressed by the register selected in the address option, added to the contents of the A register, and the result returned to the A register.

For operations in the interpretive mode operation times depended entirely upon the kind of interpretive machine specified. The basic interpretive package pretends that the AN/UYK (TRW-130) is a single-address machine with just two registers, a double-length accumulator and a double-length multiplier-quotient register. Both registers are actually locations in scratchpad memory. Now, the single precision add logram is defined as adding the operand to the low-order half of the pseudo-accumulator and replacing the low-order half of the pseudo-accumulator with the result. The execution time is 66 microseconds. The logram occupies five cells and the calling sequence two cells. The logram includes all of the interpretive control required - that is, it updates the pseudo instruction counter and transfers control to the next logram in sequence. These overhead functions require 24 of the 66 microseconds.

Typically double precision operations take quite a bit longer to execute than single precision. Because of the special facilities of the AN/UYK (TRW-130) for double precision operation, however, this rule of thumb does not hold true. The single address double precision add logram requires 9 cells and the calling sequence two cells. Again 24 microseconds are devoted to overhead.

The "special facilities" used in this double precision logram are twofold. First, the operand address in the calling sequence, which is the address of the high-order word in the operand, is automatically counted to produce the low-order word address, without loss of time. Second, the machine language add commands automatically provide for proper handling of carry and overflow for multiple precision, so no time is lost sensing and propagating a carry from least significant to most significant half.

The logram package from which these two examples have been drawn includes a full set of single and double precision arithmetic operations, data transmission operations for loading and storing the pseudo-registers, logical operations, and a full set of control and branching operations. Also included in the package are lograms for binary to BCD conversion and the reverse, and single and double precision square root, sine-cosine, arctangent and arcsine instructions. The entire package comprises 77 lograms and occupies 2056 cells.

To prepare an operational program from this package the programmer simply writes calling sequences as if he were writing symbolic instructions for another machine. A service routine assembles his program by translating the calling sequences into machine language, and also selects for loading the lograms used in his program.

Here are a few execution times that might be of interest. These are all double precision functions, yielding 29 bits in the result with an error less than $2^{-28}$:

| Function | Time (msec) |
|---|---|
| Square root | 1.7 |
| Sine-cosine | 3.8 |
| Arctangent | 2.5 |
| Arcsine | 5.9 |

The AN/UYK (TRW-130) is now fully operational. In its first customer acceptance test it logged 74 consecutive hours of operation without failure.

Footnote to the Historical Discussion:

Since preparing and delivering this paper, I have run across an internal report of Microprogramming Seminar held at M. I. T. March 1 and 2, 1956. This Seminar brought together representatives of the various streams of thought on microprogrammed computers, including Wilkes' colleague, Dr. David Wheeler; Professor Norman Scott of the University of Michigan, Lloyd Hubbard of IBM, Dr. J. J. Eachus, Doughlas Ross of M. I. T. and others who made important contributions to the concepts of microprogramming. Th full transcript of this Seminar, if it is available, is a key document in the history of the technology reported here. The fact that I did not have access to it during preparation of this paper necessarily means that the historical discussion contained here is incomplete and probably inaccurate in some respects. My only hope is that those whose contributions were not recognized will be sufficiently forgiving to await a more definitive treatment.

RHH

REFERENCES

1. Wilkes, M. V., The Best Way to Design an Automatic Calculating Machine, Manchester University Computer Inaugural Conference, Proceeding, July, 1951.

2. Wilkes, M. V., and Stringer, J. B., Micro-programming and the Design of the Control Circuits in an Electronic Digital Computer, Proceedings of the Cambridge Philosophical Society, April, 1953,

3. Paige, L. J., and Tompkins, C.B., SCAMP Postscript No. 1, Systematic Generation of Permutations on an Automatic Computer and an Application to a Problem Concerning Finite Groups; National Bureau of Standards, January, 1953.

4. Burks, Arthur W., Goldstine, Herman H., and von Neumann, Joh, Preliminary Discussion on the Design of an Electronic Computing Instrument; Institute for Advanced Study, June, 1946.

5. Glantz, H. T., A Note on Microprogramming, Journal of the Association for Computing Machinery, April, 1956.

6. Mercer, Robert J., Micro-Programming, Journal of the Association for Computing Machinery, April, 1957.

7. Gorn, Saul, Letter to the Editor, Communications of the ACM, I,1, January, 1958.