AUTOMATED DESIGN OF INDUSTRIAL SEQUENCE CONTROLLERS

R. Tabachnick*‡, T. Le-Ngoc†, L.J. Vroomen*, and P.J. Zsombor-Murray*

*DATAC Computer Laboratory, McGill University
†SPAR Aerospace Ltd.
‡The Bailey Meter Company

ABSTRACT

A system for replacing solid-state or relay based industrial control logic with custom firmware, implemented in a standard system is described. A comparison to other approaches, such as OEM implemented high level language based systems and large computer based systems is made. Existing software developments, particularly high level languages for user convenience, are discussed. The approach includes three principal concepts, viz:-

(1) A high level user-oriented programming language which allows the designer to describe an entire logic circuit with a series of keystrokes.
(2) A compiler which generates object code for a standard microprocessor based controller unit, and
(3) A microprocessor based, portable suitcase, development system which includes all the necessary hardware and firmware for field implementation and modification of sequence controllers.

The portable development system is based on the Motorola MC6800 microprocessor family while the controller uses the Motorola MC14500B 1-bit microprocessor family. The development of the keystroke language and its compiler/interpreter.

AUTOMATISATION DE LA MISE EN APPLICATION DES CONTRÔLEURS INDUSTRIELS DE MISE EN SÉQUENCE

RÉSUMÉ

Un système pour remplacer les contrôles industriels logiques à composants semi-conducteurs ou à relais par une microprogrammation adaptée aux besoins particuliers de l'utilisateur d'un système standard est décrit. L'on fait la comparaison entre les autres méthodes telles que la mise en application O.E.M. des systèmes à langage de programmation évolué et des systèmes à base de gros ordinateurs. D'autres concepts de software et particulièrement les langages évolués pour la convenance de l'usager sont traités. Les trois concepts principals de cette approche sont:

(1) Un langage de programmation évolué orienté vers l'usager qui permet un concept décrivant un circuit logique entier au moyen d'un clavier.
(2) Un compilateur produisant un code pour un contrôleur standard microprocesseur.
(3) Un système portatif à microprocesseur qui comprend tout le hardware et la microprogrammation nécessaire pour la mise en application et la modification sur le chantier des contrôleurs de mise en séquence.

La réalisation du système portatif est basée sur la famille de microprocesseurs Motorola MC6800 et le contrôleur utilise la famille de microprocesseurs Motorola MC14500B 1-bit. La mise en valeur du langage du clavier et de son compilateur/interpréteur est accentuée.

AUTOMATED DESIGN OF INDUSTRIAL SEQUENCE CONTROLLERS

by

R. Tabachnick, T. Le-Ngoc,
L.J. Vroomen and P.J. Zsombor-Murray

INTRODUCTION

Programmable sequence controllers were developed to reduce the cost involved in relay or solid state based sequence controller implementations. In order to understand the problem, consider the steps required to develop a "hardwired" sequencing system:-

1. Definition of the sequences,
2. Design of the functional circuits,
3. Fabrication of component circuit subassemblies,
4. Testing them,
5. System integration,
6. Testing the entire system and
7. Installation and startup.

Steps 1., 6. and 7. are required regardless of the technology upon which a sequence controller design is based. In programmable controllers, software writing and debugging represent steps 2., 3. and 4. By anticipating features common to the entire class of applications for which a sequence controller is intended, system integration, step 5., can be standardized to include all functions, whether they are required for a given application or not. The waste of an unused function is more than outweighed by eliminating the need to assemble and carry a variety of controller configurations. Software is the only configuration variable.

Ladder or flow diagrams are traditionally used to define sequence requirements. This has led to the evolution of logic programming languages[4,5,6]. The development of "English Language" programming languages represents efforts made to put programmable controller design within the reach of nonprogrammers[2]. These techniques severely limit the complexity of controller sequences which can be conveniently implemented. Loops which include the sequencing of several devices and which require many logic, memory and timing functions are poorly served by "English Language" descriptions. Their inadequacy stems from the inherently sequential structure of text. On the other hand, ladder diagrams, while capable of describing any sequence controller accurately, have a relatively low level of information "chunking" compared to the symbolism of multiple input logic gate circuit diagrams.

For these reasons, the customer who needs a sequence controller often insists that its functional description be in terms of a logic gate circuit diagram. The inherent adaptability of circuit diagrams to parallel structure thus makes it easy to design and comprehend relatively complicated controller networks.

The central theme of this paper is union of the logic circuit diagram, a convenient vehicle of comprehension, with the sequential structure of a stored program; inevitable if one wishes to enjoy

the flexibility and hardware standardization offered by software driven sequence controllers. In this regard, a keystroke language for translating circuit diagram logic into object code instructions for the PROM memory of a standardized hardware sequence controller is described.

STANDARDIZED HARDWARE

Consider the following prototype proposal of a standard system. This consists of a central processor board and three types of peripheral boards. Any given peripheral board may contain:-

1. Programmable timers,
2. Input buffers or
3. Output buffers, respectively.

Details of the prototype system are presented in [8] however it may be noted that boards within any given system are interconnected by a 1-bit wide data bus and a 11-bit wide address bus. Input buffers are used to convert field or process inputs into logic levels on the data bus. Output buffers convert logic level pulses on the data bus to latched, transistor driven field outputs capable of activating process device relays or status lights on an operator's console. A specific buffer bit or timer is addressable, on any board, via the address bus. Delay timing functions are performed by hardware timers because software delays are cumbersome and if many timers are required, software timing becomes entirely impractical. In the proposed system, timers can be read simply by reading an input, thus eliminating the need for interrupts. Timer hardware is described below because it is important to understand how the timing function is interfaced to software.

Timers From a logical point of view a timer is a three-state automaton. Its three states are:-

1. I ... Inactive,
2. M ... Timing and
3. X ... Finished timing, expired.

Transitions between states are controlled by the circuit input and circuit output variables:-

1. R ... Timing request, cicuit input to the timer and
2. T ... Timing flag, timer circuit output.

Fig. 1 summarizes these transitions.

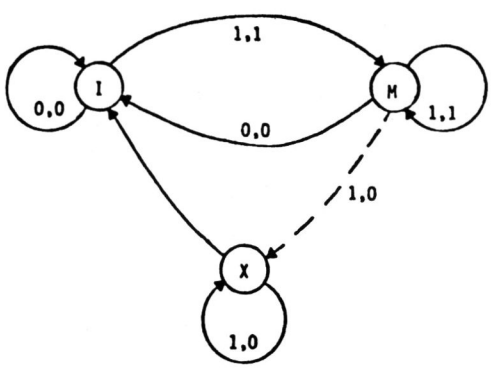


Fig. 1

Timer State Transition Diagram

From this it can be clearly seen why this function is best implemented in dedicated hardware. A classical sequential automaton, i.e., a program, cannot handle the (M)-->(X) transition without ambiguity because the circuit input R = 1 defines a transition (M)→(M), with a circuit output T = 1, while the unit is timing, as well as the timeout transition (M)-->(X) with a different circuit output, T = 0.

A timer occupies one addressable memory location, connected to three one-bit timer locations (T, R and R⊙$\overline{T}$). Two (T and R⊙$\overline{T}$) are READ-ONLY, the third (R) is WRITE-ONLY with respect to the CPU. R is accessed when the CPU is the WRITE state. Both T and R⊙$\overline{T}$ are accessed when the CPU is in the READ state, and the location is selected by the status of a FLIP-FLOP which is toggled by a preceeding software instruction.

The timer performs the following software functions:-

1. The program stores a 1 into timer location R in order to start it. This generates a 1 in the T location.
2. The program stores a 0 into timer location R. This generates a 0 in both the T and R⊙$\overline{T}$ locations.
3. The timer sets T = 0 when it has timed out. This forces the R⊙$\overline{T}$ to 1 until a 0 is stored in the R location.

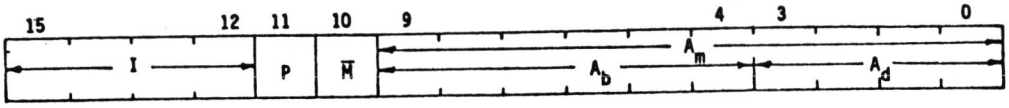
The program need not "remember" the condition of a timer. This is established, as required, by reading T and R⊙$\overline{T}$:-

	R	T	R⊙$\overline{T}$
Timer inactive	0	0	0
Impossible condition	0	1	x
Timer expired	1	0	1
Timer active, timing	1	1	0

Peripheral Boards A peripheral board contains 16 devices, all of the same type, i.e., 16 input buffers, 16 output buffers or 16 timers. A mix of up to 64 boards may be configured, with a processor board, to constitute a system. Address, data and power bus wiring is standard. Field wiring to input or output points on buffer boards is customized to suit any given controller application and its connection to customers' equipment.

Processor Board The Motorola MC14500 Industrial Control unit (ICU)[7] was selected for this system because its 1-bit wide data processing architecture is ideally suited to the logical manipulation of single bit data which constitute all the inputs and outputs. The connective transformations performed internally upon these data by any given controller implementation also result in single bit intermediate results. A wider data path, peculiar to most microprocessors, would be a disadvantage because all but one data bit would have to be tied to a common logic level in order to perform single bit operations. Otherwise, if data were stored wordwise, the processor would spend most of its computational effort in unpacking and isolating single bits.

The processor board contains EPROM for program instruction storage. A 1024 x 1-bit wide RAM is provided to store intermediate connective results and other data. Since data and instruction addressing is



Field		Width		Bits		Description
I	....	4	...	12-15	.....	Instruction (Op Code)
P	....	1	...	11	.....	Parity Bit
$\overline{M}$	....	1	...	10	.....	Input/Output Select Bit
A_m	....	10	...	0- 9	.....	RAM Memory Data Address
A_b	....	6	...	4- 9	.....	Board Select Address
A_d	....	4	...	0- 3	.....	Device Select Address

Fig. 2a
Instruction Word Format

	Instruction Code	Mnemonic	Action
#0	0000	NOPO	No change in registers. R → R, FLG0 ← ⎍
#1	0001	LD	Load Result Reg. Data → RR
#2	0010	LDC	Load Complement $\overline{\text{Data}}$ → RR
#3	0011	AND	Logical AND. RR • D → RR
#4	0100	ANDC	Logical AND Compl. RR • $\overline{D}$ → RR
#5	0101	OR	Logical OR. RR + D → RR
#6	0110	ORC	Logical OR Compl. RR + $\overline{D}$ → RR
#7	0111	XNOR	Exclusive NOR. If RR = D, RR ← 1
#8	1000	STO	Store. RR → Data Pin, Write ← 1
#9	1001	STOC	Store Compl. $\overline{RR}$ → Data Pin, Write ← 1
#A	1010	IEN	Input Enable. D → IEN Reg.
#B	1011	OEN	Output Enable. D → OEN Reg.
#C	1100	JMP	Jump. JMP Flag ← ⎍
#D	1101	RTN	Return. RTN Flag ← ⎍, Skip next inst.
#E	1110	SKZ	Skip next instruction if RR = 0
#F	1111	NOPF	No change in Registers RR → RR, FLGF ← ⎍

Fig. 2b:- MC14500B Instruction Set

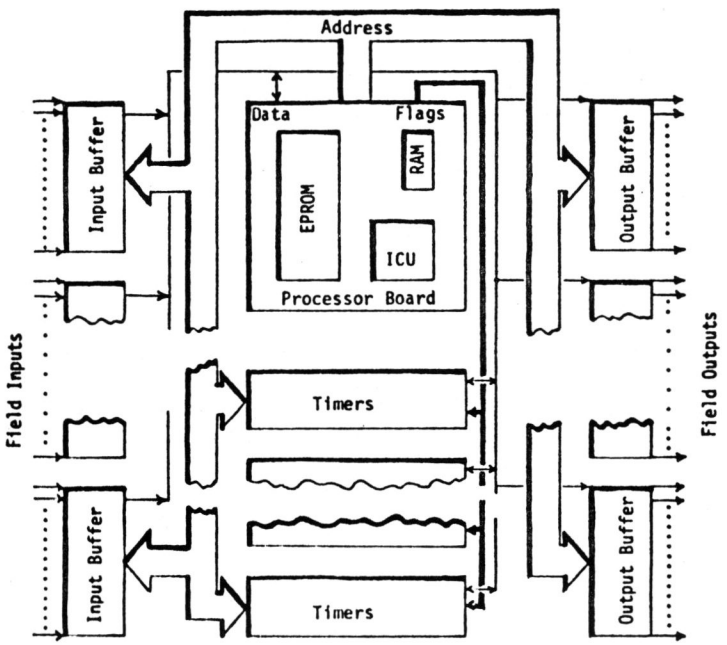


Fig. 3
System Block Diagram

performed by logic external to the ICU itself, the processor instruction word format has been tailored to suit the needs of a generalized sequence controller. This is shown in Fig. 2a. This 16-bit instruction contains a 4-bit operation code which defines the operation to be performed by the ICU during the current clock cycle; one of the 16 functions of the instruction set of the MC14500, Fig. 2b. Aside from the error checking parity bit, P, the rest of the word is an address operand which specifies one of 1024 RAM memory bits if $\overline{M} = 0$. If $\overline{M} = 1$, one of the 16 devices, specified by A_d, is addressed on one of the peripheral boards, specified by A_b.

An integrated system block diagram is shown in Fig. 3. The bus interconnection among the modules carries not only the data and address paths, but two processor flag signal lines as well. These serve to access timer functions.

SOFTWARE

Programming Language The programming language was designed on the assumption that the sequence controller to be implemented was designed as a gate type logic circuit. The language is structured such that logic elements and their interconnections, e.g., the circuit shown in Fig. 4, are implemented by a sequence of keystrokes, on the keyboard, Fig. 5.

Fig. 6 specifies the Backus Normal Form (BNF) of the keyboard based, circuit specification language grammar. Figs. 7a through 7k show in detail how the circuit of Fig. 4 is parsed into 11 subsequences or records.

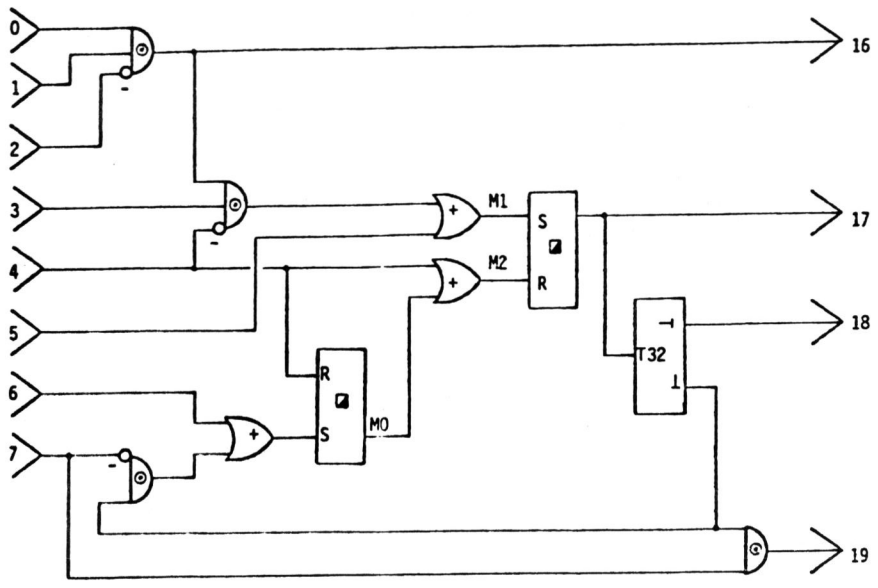
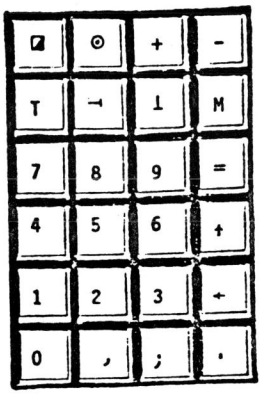


Fig. 4
Example Sequence Controller Circuit



Key	Description
◪	R-S Flip-Flop
⊙	And-Gate
+	Or-Gate
-	Negate signal
T	Input starts timer
⊣	Output true if timing
⊥	Output true if timer expired
M	Temporary storage in RAM
=	Assign signal unconditionally
0-9	Signal/function enumerators
,	Input signal separator
;	Store/recall previous result for next input operand
.	End of line/enter
↑	Delete to previous end of line
←	Delete previous symbol

Fig. 5

Programming Keyboard

< *RECALL OPERAND* > :: = ;

< *OPERAND* > :: = 0|1|2|3|4|5|6|7|8|9|< *OPERAND* >< *OPERAND* >

< *OPERATOR* > :: = ◪|⊙|+|T|⊣|⊥|=

< *TERM* > :: = < *OPERAND* >,< *OPERAND* >|-< *OPERAND* >|M< *OPERAND* >|< *OPERAND* >,< *TERM* >

< *EXPRESSION* > :: = < *TERM* >< *OPERATOR* >|;< *TERM* >< *OPERATOR* >

< *INSTRUCTION* > :: = < *EXPRESSION* >< *OPERAND* >.|< *EXPRESSION* >;|< *EXPRESSION* >M< *OPERAND* >.|
< *EXPRESSION* >< *OPERAND* >;

< *PROCEDURE* > :: = < *INSTRUCTION* >.|< *INSTRUCTION* >< *INSTRUCTION* >.

Fig. 6

B N F Grammar of Keystroke Language

Note that multiple input operands are separated by (,) and records are terminated by (.). A record is defined by a sequence which results in an output. There are four types of record output:-

1. A field output connected to the customer's equipment,
2. An output which starts a timer,
3. An output which, though not assigned to the field or to a timer, will implicitly constitute the first input in the next record and
4. An output, such as 3. above, which cannot logically be used in the next record and hence must be stored temporarily so that it may be referenced explicitly, later, when it is required as an input.

Sample Program

```
0,1,-2 ⊙16.
;3,-4 ⊙;
5+M1.
4,M0+M2.
M1,M2 ◪17.
;T32.
-⊙32=18.
⊥32,7 ⊙19.
-7,⊥32 ⊙ ;
6+;
4 ◪M0..
```

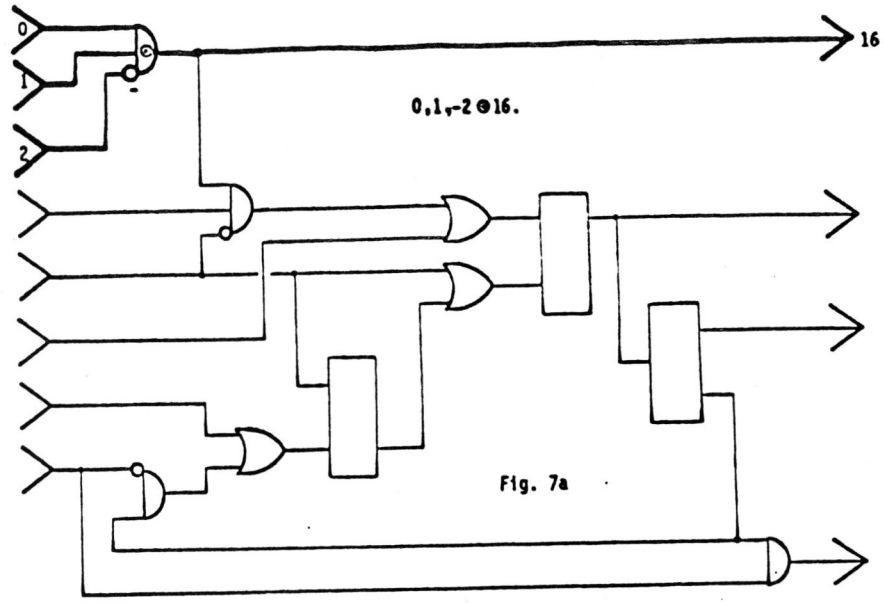


Fig. 7a

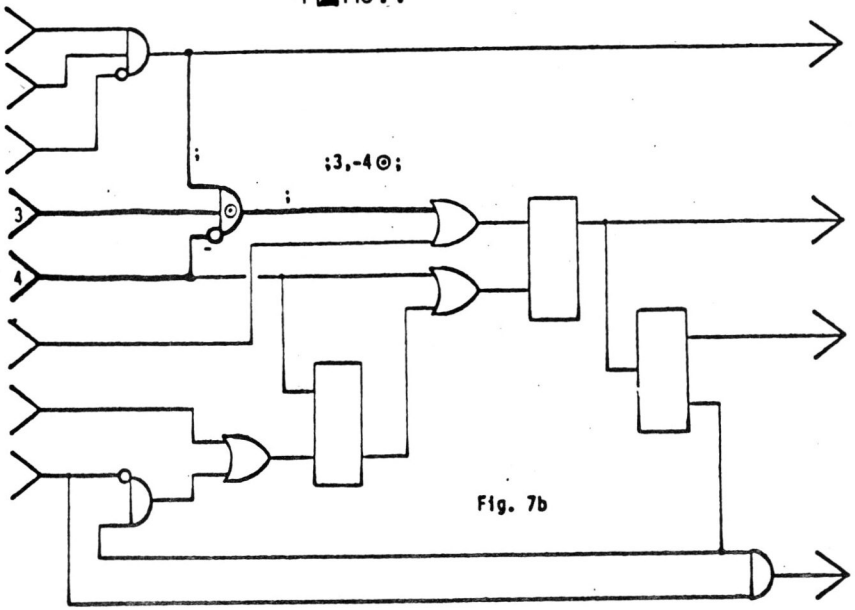


Fig. 7b

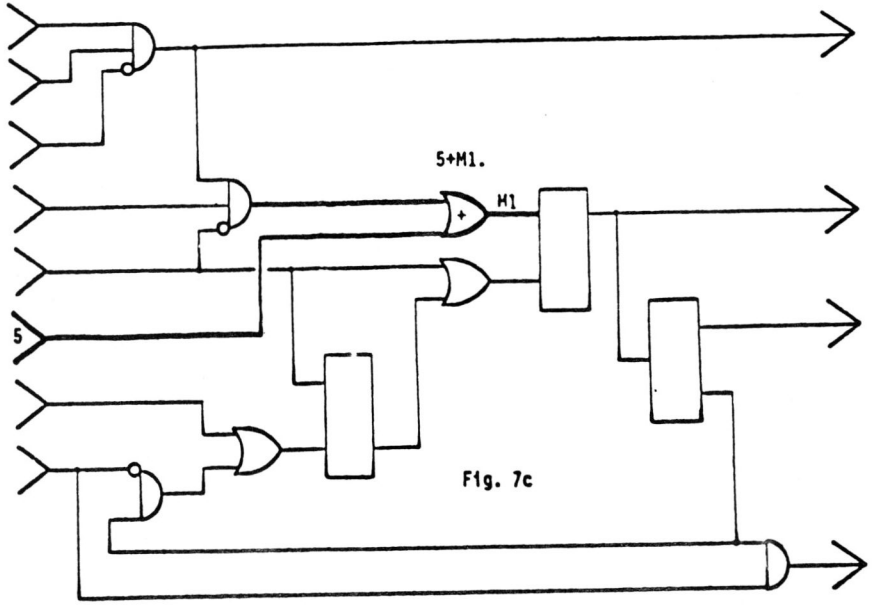
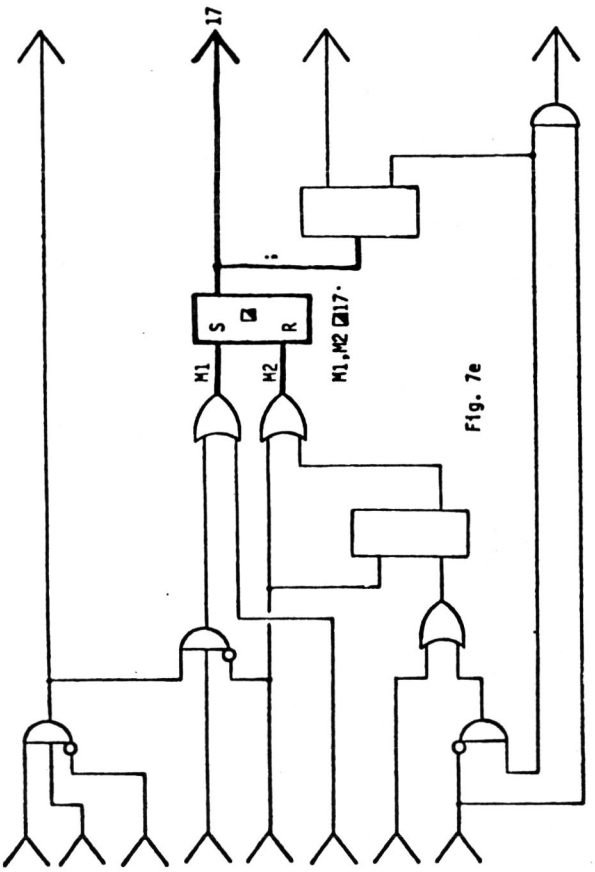
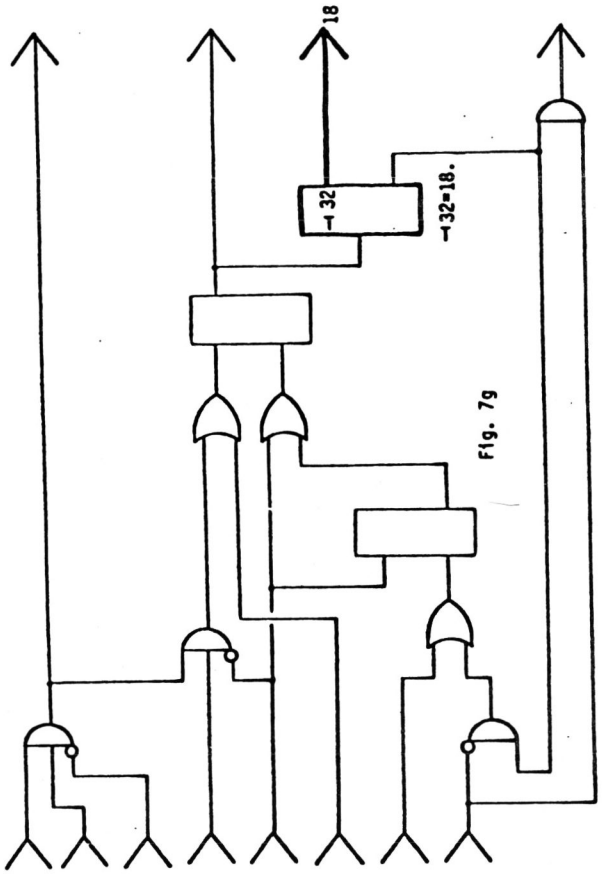
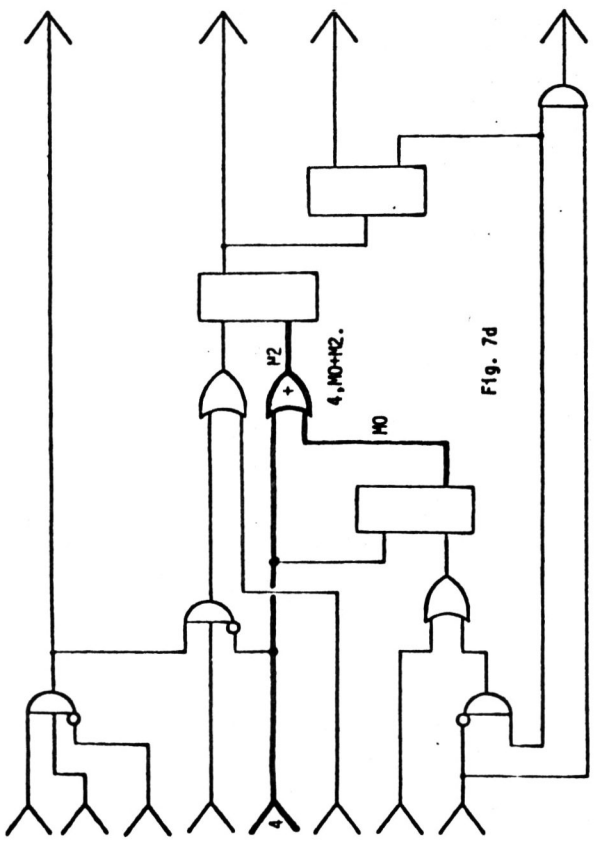
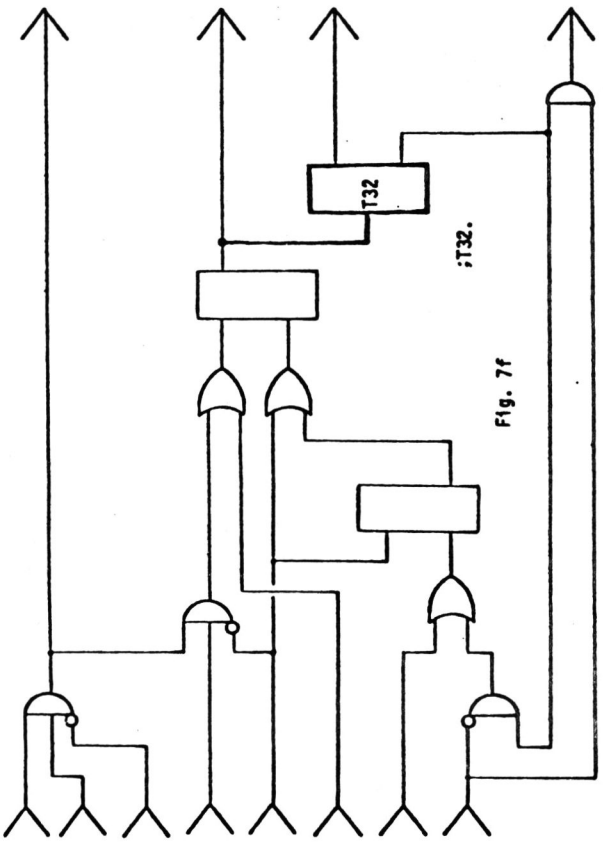


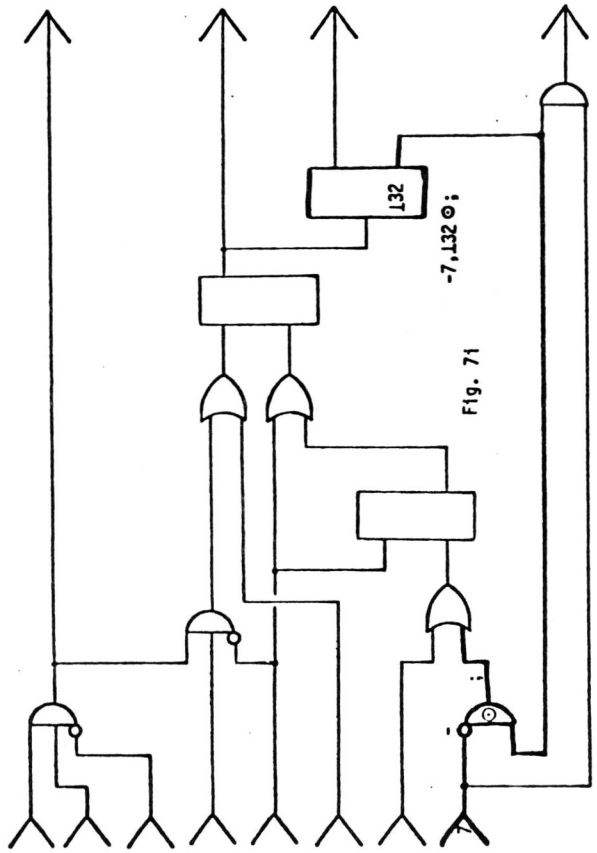
Fig. 7c

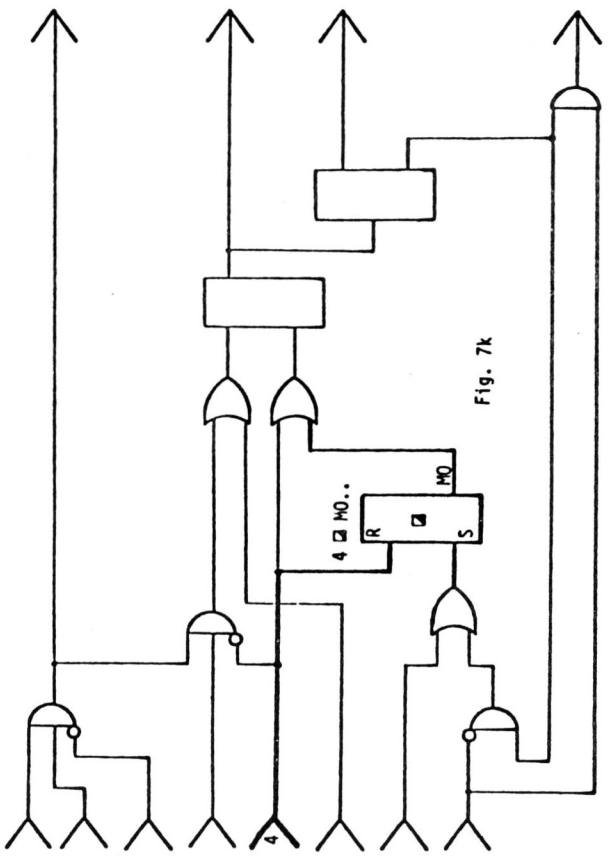

17
S
R
M1
M2
M1,M2 ⊡17.
;
Fig. 7e

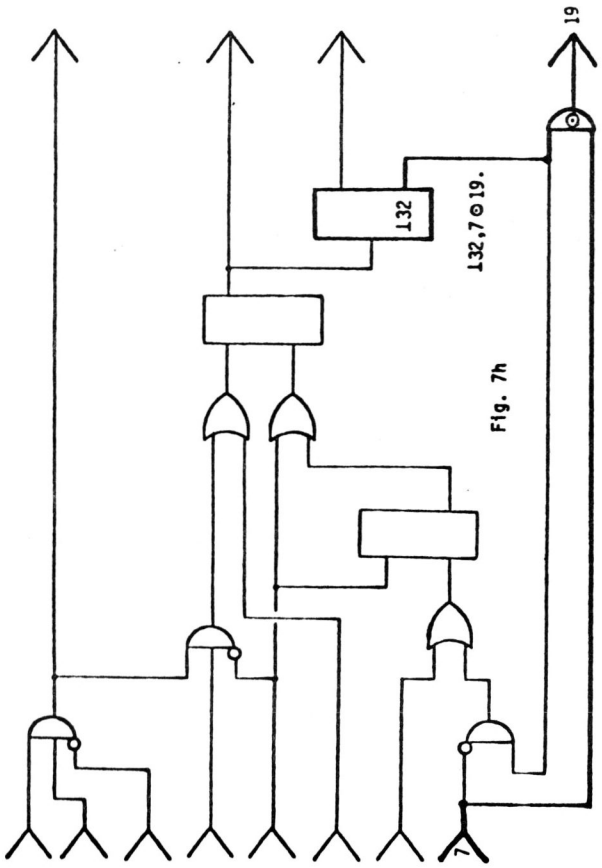

18
-1 32
-32=18.
Fig. 7g

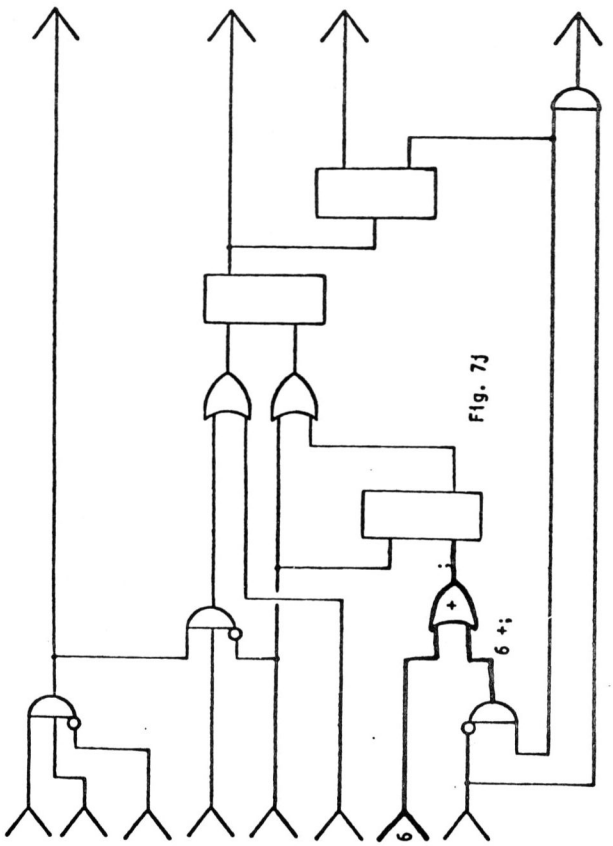

M2
+
4,M0+M2.
M0
4
Fig. 7d


T32
;T32.
Fig. 7f


⊥32
-7,⊥32 ⊙;
Fig. 7i


4 ◘ MO..
MO
R
S
Fig. 7k


19
⊥32
⊥32,7 ⊙ 19.
Fig. 7h


6 +;
Fig. 7j

A record closed with a (,) must have an explicit output such as:-

1. Field output:- (... 17,),
2. Start timer:- (... T32,) or
3. Temporarily assigned to memory:- (... M1,).

If not, it will be lost unless recalled by an initial, opening (;) in the next record. Conversely, a record closed with a (;) will regenerate the implied output as the implicit, first operand of the next record. Proper use of (;) will enhance the object program by inhibiting the generation of extra LD and/or STØ instructions which would inevitably result if the output, and its use as a subsequent input, were explicitly specified. E.g., the sequence:-

```
0,1,-2 ⊙16,
;3,-4 ⊙;
```

saves the extra LD which would be generated by the second record of the following, logically identical, explicit sequence:-

```
0,1,-2 ⊙16,
16,3,-4 ⊙;
```

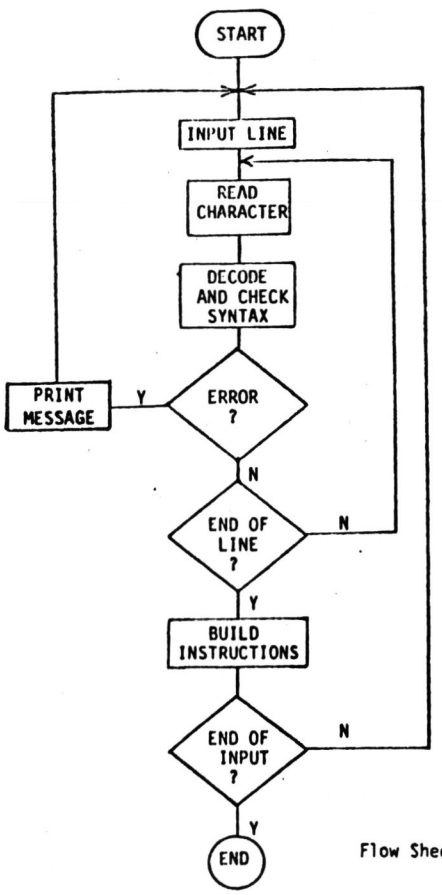


Fig. 8

Compiler Flow Chart

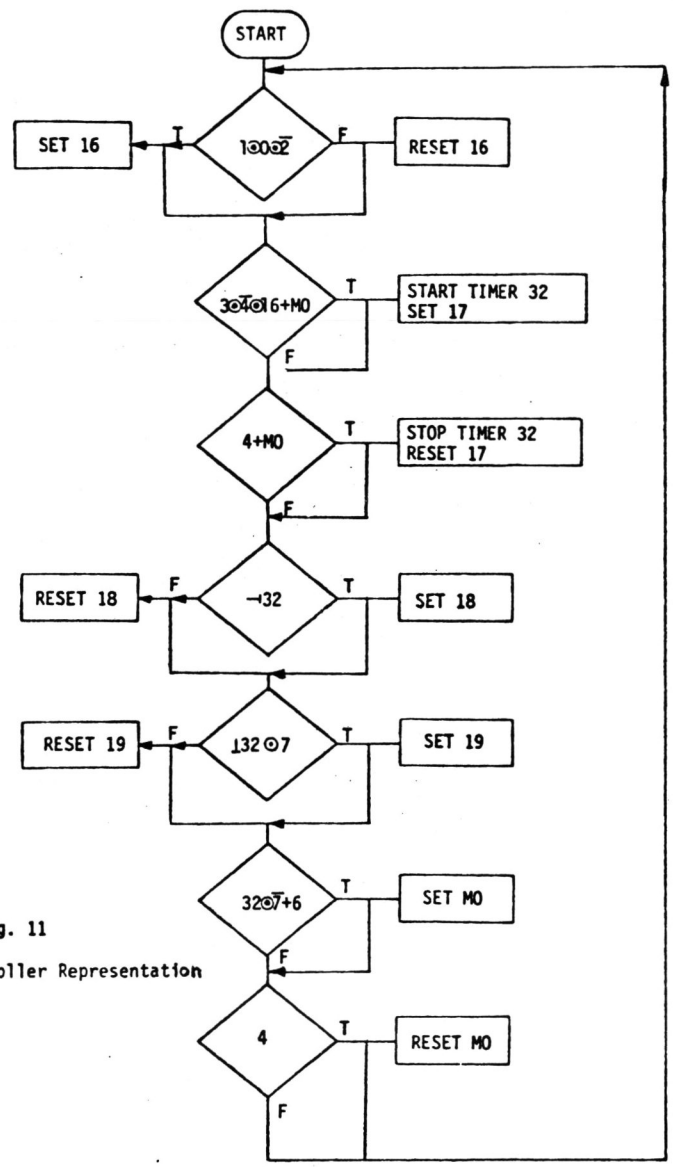


Fig. 11

Flow Sheet Controller Representation

The start timer input, (T32), is a grammatical output which disappears into the timer function. Similarly, timer outputs, (⊣32) and (⊥32), are grammatical inputs, to other circuit elements, which originate in the timer function, e.g., (⊣32=18,). It should be noted, furthermore, that the first input operand to a flip-flop, (◪), is the "set", S, input. The second is the "reset", R, input, e.g.:-

M1,M2 ◪17,

Compiler A flow chart outlining the logic of the compiler, which implements a controller circuit specified by a keystroke sequence by translating this into object code for the MC14500, is shown in Fig. 8. Currently, this compiler, composed in FORTRAN IV, runs on a HS4020 system and uses a typer keyboard instead of the special keypad of Fig. 5. The equivalent mnemonics, the typer input, the MC14500 symbolic assembler and the resulting binary object code that it generates are illustrated in Fig. 9.

COMPARISONS

The example, Fig. 4, has been programmed using procedures outlined in[5] and in flow chart form, but not strictly in accordance with the procedure outlined in[4], respectively. The reader is invited to compare the proposed logic circuit/keystroke system to the roughly equivalent software implementations using:-

1. A ladder diagram based design approach, Fig. 10, and
2. One based on a flow sheet representation, Fig. 11, of the sequence controller.

Relay Ladder Diagram Fig. 10 shows the relay ladder diagram equivalent of the sequence controller circuit in Fig. 4. The number of keystrokes required for the programmable matrix controller implementation is only slightly greater than the number of lines of assembly code produced by the compiler and is about the same as the number of keystrokes shown in Fig. 7. It must be pointed out, however, that the Allen-Bradley system requires that input/output addresses be entered separately, via thumbwheel switches. Keystrokes are used for operation codes only. There are no operand keys. No special treatment is bestowed upon timers and this may be seen as an advantage or otherwise, depending on the designer's preference.

It seems that ladder diagrams suffer from a shortcoming which tends to obscure the controller function. The contacts of any given relay, except in the case of a self-latching relay, are more or less equivalent to the inputs of logic function whose output energizes the coil of *another* relay. Simple, multiple input logic functions are spread all over the ladder map. Even for those familiar with and partial to ladder diagram design, it must be difficult to use a tool which distributes, rather than concentrates, interrelationship. Furthermore, controller inputs are not easily identifiable. One must search for contacts which lack a solenoid.

Flow Sheet Implementation A translation of Fig. 4 into a flow sheet is shown in Fig. 11. It is evident that controller functions, even for this simple configuration, are obscured by piecemeal decomposi-

tion into auxiliary fragments. In this regard it was not obvious from descriptions given in[4] how a "Versatyme" controller handles a timer function if it is required to check timer status, i.e., (⊣) or (⊥).

EQUIVALENT MNEMONICS

Teletype Keyboard		Programming Keyboard
A	.....	⊙
O	.....	+
X	.....	−
B	.....	T
T	.....	⊣
E	.....	⊥
,	.....	,
M	.....	;
(carriage return)	.....	'
F	.....	◪
L	.....	M
>	.....	=

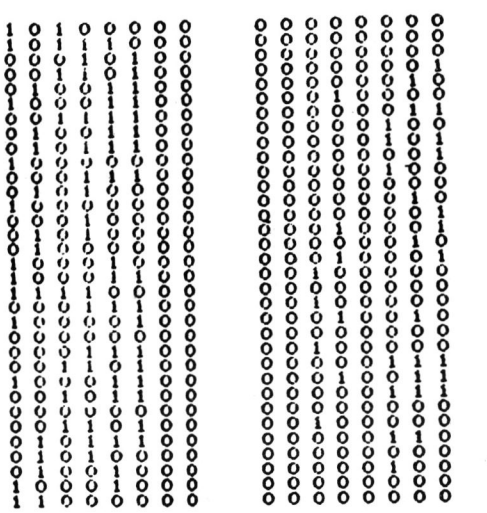
Teletype Keyboard Input

```
M14500
USER?10106
0,1,X2A16
M,3,X4AM
50L1
4,L00L2
L1,L2F17
MB32
T32>18
E32,7A19
X7,E32AM
60M
4FL0
```

Fig. 9
Using the Compiler

Compiler Output

binary

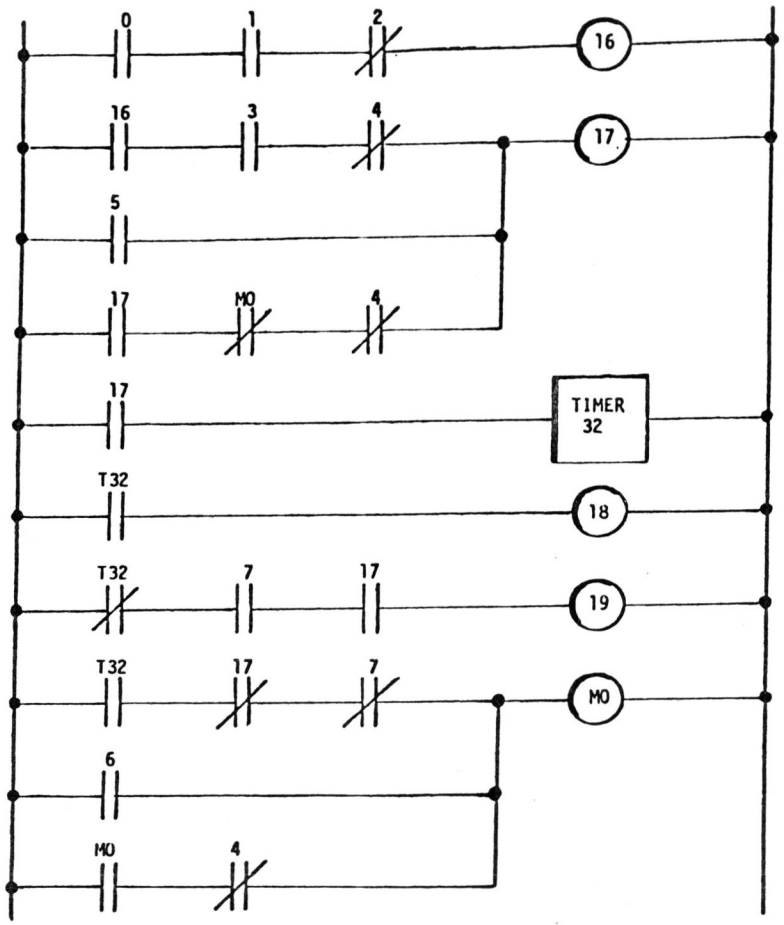


hexadecimal assembler

```
A0 00
B8 00
14 00
34 01
4C 02
8C 10
3C 03
4C 04
5C 05
80 01
1C 04
50 00
80 02
10 01
50 11
40 02
88 11
8C 20
F0 00
1C 20
84 12
00 00
1C 20
34 07
8C 13
2C 07
00 00
34 20
5C 06
50 00
48 04
88 00
C0 00
```

assembler

```
1EN      0
OEN      0
LD    1024
AND   1025
ANDC  1026
STO   1040
AND   1027
ANDC  1028
OR    1029
STO      1
LD    1028
OR       0
STO      2
LD       1
OR      17
ANDC     2
STO     17
STO   1056
NOPF     0
LD    1056
STO   1042
NOPO     0
LD    1056
AND   1031
STO   1043
LDC   1031
NOPO     0
AND   1056
OR    1030
OR       0
ANDC     4
STO      0
JMP      0
```




```
XIC 0          XIC 5          XIC 32         XIØ 6
XIC 1          BRT            SET 18         BRT
XIØ 2          XIC 17         XIØ 32         XIC 7
SET 16         XIØ 4          XIC 17         BRT
XIC 16         XIC MO         XIC 6          XIC MO
XIC 3          SET 17         SET 19         XIØ 4
XIØ 4          XIC 17         XIC 17         SET MO
BRT            SET 32         XIØ 32
```

XIC ... *Normally Open* contact
XIØ ... *Normally Closed* contact
BRT ... *OR*
SET ... Energize output

Fig. 10

Programmable Matrix Controller Implementation of Ladder Diagram Design

CONCLUSION

The development of high level languages and complementary system development hardware serves to reinforce the advantages which microprocessor based sequence controllers enjoy with respect to their hardwired counterparts. The logic circuit/keystroke approach described herein should yield a substantial improvement in the productivity of sequence controller designers; especially those who have some background in digital logic and assembly language programming.

An impromptu competition, with an interesting outcome, was held. Five sequence controller designer/programmers set about the implementation of a simple circuit based design; one which could be conveniently evaluated on a commercially available MC14500 ICU evaluation kit. The programmers had varying degrees of experience. One used the preliminary, HS4020 based compiler while the others confined themselves to hand coding in MC14500 assembly code. The computer assisted implementation tied for first place in terms of the number of instructions required to correctly emulate the target control circuit. For obvious reasons, the speed with which the five contestants accomplished their task will not be compared.

ACKNOWLEDGEMENT

This research is supported by Natural Sciences and Engineering Research Council Canada grant number A4219 and the Bailey Meter Company Limited of Canada.

REFERENCES

[1] Maggioli, V.J., "How to Apply Programmable Controllers", Hydrocarbon Processing, Vol. 57, No. 12, Dec. 1978, pp. 137-142.

[2] Smith, G.H., "Converting Relay Logic to Software", Machine Design, Vol. 50, No. 19, 24 Aug. 1978, pp. 93-99.

[3] Sanderson, L.B. and Lord, J.C., "Microcomputers Promise Less Stop, More Go", IEEE Spectrum, Vol. 15, No. 11, Nov. 1978, pp.30-32.

[4] Versatyme System 2 Programmable Controller, Product Specification, Versatyme Controls Corporation, 1333 Lawrence Expressway, Suite 360, Santa Clara, CA 95051, 1978.

[5] Allen-Bradley Publication SD23, Allen-Bradley Systems Div., Highland Heights, OH 44143, Apr. 1971.

[6] Togino, K. and Fukura, K., "A Computer Route to Ladder Diagrams and PC Programs", Instrumentation Technology, Vol. 25, No. 5, Sept. 1978, pp. 125-130.

[7] Gregory, V., Dellande, B. *et al*, "MC14500B Industrial Control Unit Handbook", Motorola Semiconductor Products Inc., 1977.

[8] Tabachnick, R.L. and Zsombor-Murray, P.J., "An Approach to the Implementation of Industrial Sequence Controllers with Standardized Hardware and Firmware", Proc. Cdn. Conf. on Auto. Control, McGill University, Montreal, QU H3A 2K6, 23-25 May 1979. (to be published)