intel®

ARTICLE REPRINT

AR-17

What can you do with a microprocessor?

Robert H. Cushman
Special Features Editor, EDN

EDN Magazine, March 20, 1974

What can you do with a microprocessor?

Here is how you can take $50 worth of microprocessor parts and control a real-world electro-mechanical system.

Robert H. Cushman, Special Features Editor

By now you are probably sick and tired of hearing all these endless, vague claims about how the microprocessor will replace all TTL and CMOS SSI and MSI circuits and make you, the digital system designer, obsolete in the process. You've had your fill of our dissertations on the architectures of these μP's and how they condense roomfuls of 1960-vintage computers and rackfuls of 1970-vintage mini-computers into a handful of DIP's.

You say, "Just give me one quick, simple and to-the-point example of how a microprocessor can replace TTL in the small systems I am designing—systems where the end products sell for no more than a thousand dollars or so. Tell me how I can get in the μP act."

We have a double-barreled answer for you: Intel's little MCS-4 μP, and Matt Biewer of Pro-Log Corp., Monterey, Calif. Together, they are the basis for the common-sense application example that follows—where a μP replaces the discrete circuitry that normally operates a paper-tape reader.

Our choice of the MCS-4 was because it's one of the very few microprocessors whose parts are actually on distributors' shelves. (We know because we've called them.) Further, the MCS-4 is priced so low that it can do the example for $50, which is roughly equivalent to the cost of commercial TTL cards for tape readers. The MCS-4, of course, can do much more than the example. In the real application it comes from, this is just a minor subroutine.

We've chosen Matt Biewer because he seems to us a sort of circuit designer's μP "folk hero." He's already gone the route you will be going as you add microprocessors to your bag of tricks. He has self-taught himself programming and developed a no-nonsense hexadecimal-code approach that has allowed him to detour around the need to use assembler and compiler programs.

However, many μP systems' planners are rushing on to 8-bit machines and many μP software experts sharply disagree with Biewer's humble "up-from-hardware" approach. An Intel spokesman who knows Biewer personally, while admitting that Biewer gets results, argues that to follow his footsteps is to ignore a decade of software development.

Our conclusion after hearing both sides of the story is that, for the small microprocessors like the MCS-4 which are suitable for $50 controllers, the Biewer manual approach makes sense for circuit designers first getting into μP's. We agree

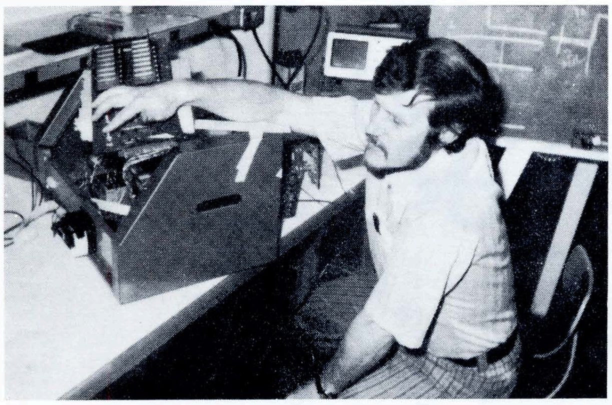


Fig. 1—Matt Biewer, creator of the tape-reader design example, is an ex logic-circuit designer who has switched over to μP's. He got started with microprocessors while with a former employer, MSI-Data, by designing the MCS-4 into a portable terminal instead of the custom LSI originally considered. Hundreds of units of that product are now rolling off the production line each month. Biewer is now vice-president with Pro-Log Corp., Monterey, Calif., a small company that sells support products for μP's.

whole-heartedly with Biewer that it may be too much to ask a circuit designer to learn both μP hardware and the concepts of higher level language programming at the same time. Besides, you just can't afford that much ROM memory in a $50 system that the programming can't be done manually, as Biewer does. But we think that it might be very wise for the designer to be learning about higher level language programming at night school while he works on the Biewer machine-level approach by day.

At this time we won't get into the argument whether Biewer is right or wrong to circumvent the niceties of the world of software. The point to be made here is that the program you see in the example was created and entered into the pROM programmer by hand, without the aid of an assembly program, by an ex-digital circuit designer like yourself.

Practically no interface hardware

The hardware side of Biewer's tape reader control is shown in **Fig. 2**. The MCS-4 part is essentially just a 4004 CPU and three 4001 ROM's with their I/O ports, all driven by a 750 kHz 2-phase clock. Or at least this is what the MCS-4 part of the system would look like if the product were in high-volume production. Biewer's application is in initial moderate-volume stage, and he replaces the custom 4001 metal-masked ROM's shown in **Fig. 2** with the easier-to-program Intel 1702A ROM's. He interfaces these to the MCS-4 with the 4008/4009 chips that Intel makes for this purpose and adds TTL latches and buffers for the I/O ports. But in **Fig. 2**, to keep our diagram simple, we show the same 4001 ROM's that we described in our last article (**Ref. 5**).

Note from **Fig. 2** how little external hardware is needed to interface the tape-reader stepping motor and hole sensors to the MCS-4 I/O ports. The buffer amplifiers that boost the output of the MCS-4 port 6 to the level adequate for the motor coils can be very simple. We show them as just IC Darlingtons with built-in protective diodes. They would have to be there anyway—you can't blame them on the MCS-4.

We, likewise, show the photo-transistor hole sensors as high-gain Darlington types to minimize parts count. So you can see that the only additional hardware elements to interface this typical electro-mechanical function to the existing MCS-4 system are the three I/O ports 4, 5 and 6. With the standard MCS-4 parts, this means you have to have at least three 4001 ROM's. If you look ahead to the small number of lines in the program, **Fig. 3b**, you might say your I/O requirements will force you to put in more ROM than the application needs. (Each ROM in the MCS-4 system has 256 lines, and there are only 37 lines in the software program.) However, there are many additional functions that you can implement with the spare software space in a system like this. The chances are you will find that even if your main program doesn't need the ROM space—which it most likely will—there are valuable uses for the additional ROM lines in embellishments of this tape-reading function. Don't worry, you'll think of them.

What it takes to move tape

The software program, **Fig. 3**, has to be "the design" because you've already seen how little hardware there is from **Fig. 2**. The flow diagram, **Fig. 3a**, is the programmer's conceptual starting point in the process of "verbalizing a program," to use Matt's words. It is the skeleton which is clothed with the flesh of the instruction sequence of **Fig. 3b**.

This flow diagram shows two entry points for the subroutine, ROM address 67 for STEP FORWARD AND READ TAPE and ROM address 60 for STEP REVERSE AND READ TAPE. It shows one exit point, address 7F (hexadecimal) at the bottom where the BBL instruction at the end of this subroutine returns the computer to the main program. (See brief list of instructions and their explanations in box in **Fig. 3**.)

The entry points, 60 and 67, would be reached by a JMS or "jump to subroutine" at any point in the main program where it was desired to read in more data from the tape. That main-line JMS instruction would put either a 60 or 67 in the program counter (PC), which would push the previous main-line program address down in the PC stack (to be available for later return at the final BBL). This would cause the CPU to fetch the first instruction of this subroutine. (The MCS-4's basic machine cycle was explained in our last article, **Ref. 5**.)

The approximately seven parts to this software program are depicted by the blocks and diamonds of the flow chart, **Fig. 3a**. We've separated the lines of code into groups to correspond with these blocks.

The purpose of the first seven code lines is to take the two side-by-side ONEs in the 4-bit pattern that will drive the four stepper-motor coils and shift the two side-by-side ONEs right or left to rotate the motor and move the tape for the new reading position. This pattern has been stored in CPU index register A (hexadecimal 10 in decimal). It was put there after the last rotation (using the instruction at 71 as we'll see). Now it has to be brought out into the accumulator, rotated left or right according to the desired motion of the motor, and presented to the output

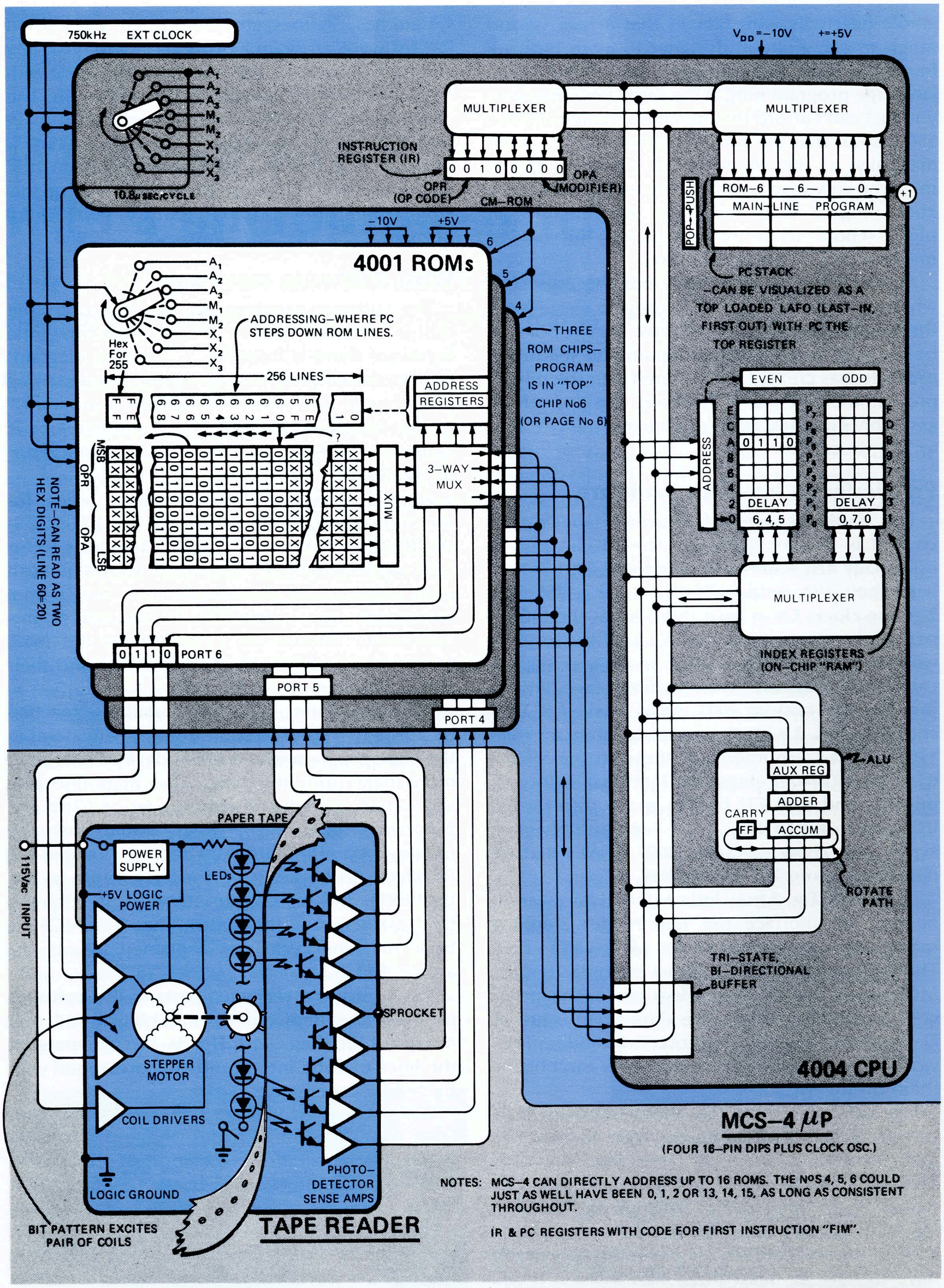
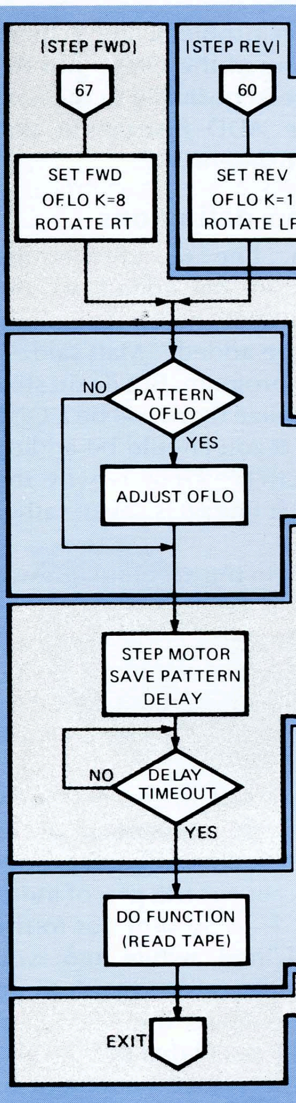

750kHz EXT CLOCK
10.8μ sec/cycle
A1 A2 A3 M1 M2 X1 X2 X3
MULTIPLEXER
INSTRUCTION REGISTER (IR)
0 0 1 0 0 0 0 0
OPR (OP CODE)
OPA (MODIFIER)
CM–ROM
VDD=–10V +=+5V
MULTIPLEXER
POP–PUSH
ROM–6 —6— —0—
MAIN–LINE PROGRAM
+1
PC STACK –CAN BE VISUALIZED AS A TOP LOADED LAFO (LAST–IN, FIRST OUT) WITH PC THE TOP REGISTER
–10V +5V
6 5 4
4001 ROMs
Hex For 255
ADDRESSING—WHERE PC STEPS DOWN ROM LINES.
256 LINES
ADDRESS REGISTERS
MSB OPR OPA LSB
MUX
3–WAY MUX
THREE ROM CHIPS– PROGRAM IS IN "TOP" CHIP No6 (OR PAGE No 6)
NOTE—CAN READ AS TWO HEX DIGITS (LINE 60=20)
0 1 1 0 PORT 6
PORT 5
PORT 4
EVEN ODD
ADDRESS
0 1 1 0
DELAY 6, 4, 5
DELAY 0, 7, 0
MULTIPLEXER
INDEX REGISTERS (ON–CHIP "RAM")
AUX REG
ADDER
CARRY FF
ACCUM
Σ–ALU
ROTATE PATH
TRI–STATE, BI–DIRECTIONAL BUFFER
4004 CPU
PAPER TAPE
115Vac INPUT
POWER SUPPLY
+5V LOGIC POWER
LEDs
SPROCKET
STEPPER MOTOR
COIL DRIVERS
LOGIC GROUND
PHOTO–DETECTOR SENSE AMPS
BIT PATTERN EXCITES PAIR OF COILS
TAPE READER
MCS–4 μP
(FOUR 16–PIN DIPS PLUS CLOCK OSC.)
NOTES: MCS–4 CAN DIRECTLY ADDRESS UP TO 16 ROMS. THE NOS 4, 5, 6 COULD JUST AS WELL HAVE BEEN 0, 1, 2 OR 13, 14, 15, AS LONG AS CONSISTENT THROUGHOUT.
IR & PC REGISTERS WITH CODE FOR FIRST INSTRUCTION "FIM".

Fig. 2—The hardware side of the tape reader control: MCS-4 parts are shown at the top in much the same "classical" arrangement as they were in the preceding article (**Ref. 5**). The tape reader peripheral is shown at the bottom. As explained in the text, Biewer actually uses a 1702A electrically-programmable,U V light-erasable pROM instead of the regular 4001 ROM that is part of the MCS-4 family. He interfaces the 1702A with the 4008 and 4009 MOS/LSI parts that Intel makes for the purpose. But he will go to this configuration, once his product reaches volume production.



A) FLOW DIAGRAM

ADR HEX	INSTR HEX	LABEL	MNEMONIC	OPERAND	COMMENTS
60	20	(STEP REV)	FIM	PO	SET OUTPUT ADDRESS &
61	60		6	O	OVERFLOW CONSTANT (0+CARRY=1)
62	F1		CLC		ROTATE THE STEP PATTERN
63	AA		LD	A	(REG A) $\longrightarrow$ ACCUMULATOR
64	F5		RAL		ROTATE LEFT
65	43		JUN		
66	6C			CHK OFLO	
67	20	(STEP FWD)	FIM	PO	SET OUTPUT ADDRESS &
68	67		6	7	OVERFLOW CONSTANT (7+CARRY=8)
69	F1		CLC		ROTATE THE STEP PATTERN
6A	AA		LD	A	
6B	F6		RAR		ROTATE RIGHT
6C	1A	CHK OFLO	JCN	CO	TEST FOR OVERFLOW
6D	6F			NO CARRY	
6E	81		ADD	1	ADJUST FOR OVERFLOW
6F	21	NO CARRY	SRC	PO	ADDRESS THE OUTPUT LINES
70	E2		WRR		SEND NEW PATTERN TO STEP MOTOR
71	BA		XCH	A	SAVE THE NEW PATTERN
72	22		FIM	PI	DELAY LONG ENOUGH TO ALLOW THE
73					MOTOR TO STEP AND STOP IN IT'S
74	56		JMS		NEW POSITION (≈10 MSEC FOR A
75	80			DELAY	100 CPS READER)
76	20		FIM	PO	ADDRESS INPUT LINES TO READ
77	40		4	0	THE LEAST SIG. BITS.
78	21		SRC	PO	
79	EA		RDR		READ THE LEAST SIG. BITS (LSB)
7A	B6		XCH	6	SAVE THE LSB
7B	60		INC	0	ADDRESS INPUT LINES TO READ
7C	21		SRC	PO	THE MOST SIG. BITS.
7D	EA		RDR		READ THE MOST SIG. BITS (MSB)
7E	B7		XCH	7	SAVE THE MSB
7F	CO		BBL	0	"POP" PC STACK FOR RETURN
80	72	DELAY	ISZ	2	COUNT REGISTERS TO ZERO
81	80		80	0	
82	73		ISZ	3	
83	80		80	0	
84	CO		BBL	0	RETURN

B) SEQUENCE OF INSTRUCTIONS (PROGRAM)

(C) List of MCS–4 instructions used in the example*

Mnemonic OPR	Mnemonic OPA	Description	Number of ROM lines
FIM	Reg. Pair (4 bits+4 bits)	FETCH IMMEDIATE Load the eight bits of data from the following ROM line into the designated register pair. Source is next ROM line and destination is two registers in CPU.	2
CLC		CLEAR CARRY Set the carry to zero.	1
LD	Register	LOAD REGISTER TO ACCUMULATOR The 4–bit content of the designated CPU register is loaded into the accumulator.	1
RAL		ROTATE LEFT The content of the accumulator is rotated through the carry to the left one bit position.	1
JUN	A_3 (A_2, A_1)	JUMP UNCONDITIONAL The ROM memory address A_3, A_2, A_1 is loaded into the CPU program counter (PC) so that next instruction is fetched from ROM location, A_3, A_2, A_1.	2
RAR		ROTATE RIGHT (See rotate left above.)	1
JCN	Condition (A_2, A_1)	JUMP ON CONDITION If the designated condition is true, the address A_2, A_1 is put into the CPU program counter (PC). Sixteen different tests are possible.	2
ADD	Register	ADD REGISTER TO ACCUMULATOR The 4–bit content of designated CPU register is added to the contents of the accumulator with carry, with results stored in accumulator.	1
SCR	Reg. Pair	SEND REGISTER CONTROL Send contents of designated index register pair to the I/O ports as chip select. (Also for addressing RAM locations when RAM chips present.)	1
WRR		WRITE ROM PORT Content of accumulator is transferred to output port previously selected by SRC instruction. This data is available at the output pins until a new WRR is executed on same port.	1
XCH	Register	EXCHANGE REGISTER WITH ACCUMULATOR The 4–bit content of the designated CPU register is loaded into the accumulator and the content of the accumulator is loaded into the register.	1
JMS	A_3 (A_2, A_1)	JUMP TO SUBROUTINE The address A_3, A_2, A_1 is pushed onto the stop of the program counter (PC) stack. The previous content of the PC is pushed down one level to be saved for return from subroutine.	2
RDR		READ ROM PORT Data present at the port previously selected by an SRC instruction is transferred to the accumulator.	1
INC	Register	INCREMENT REGISTER The 4–bit content of the designated register is incremented by one.	1
BBL		BRANCH BACK AND LOAD ACCUMULATOR The PC program counter stack is popped up one level. This causes PC to return to where it left off in previous program before the JMS occurred.	1
ISZ	Register (A_2, A_1)	INCREMENT REGISTER, SKIP IF ZERO Contents of designated register are incremented by one. If the result is zero, the PC continues down the ROM instruction lines in sequence. If the result is not zero, the PC goes to address line A_2, A_1.	2

*Note: These are but 16 of the MCS–4's 46 total instructions.

Fig. 3—Software for the tape-reader control: this is what was developed to go into the MCS-4 ROM so that the µP would perform the control actions. He first sketched out his program graphically in the flow diagram (**a**) and with this to guide him, he selected the sequence of MCS-4 instructions that implemented the flow diagram (**b**). A capsule explanation of the MCS-4 instructions used is given in the box (**c**).

port leading to the motor coils and then, again, be put back into A for the next time.

How software moves an actuator

Suppose the STEP REVERSE command leg is entered by the JMS from the main program. The FIM instruction at 60 is a two liner taking up two ROM lines (each having its own full machine cycle). It says register pair No. 0 (which is index register 0 and 1, if you look back to **Fig. 2**) will be loaded with the constant in the second line—address 61. The "6" in the left 4-bit side of this second line will be used shortly in step 6F to address the output port leading to the motor. The ZERO on the right 4-bit side, is a bit tricky. Ordinarily, it would not matter what this was when an I/O port was being addressed, for you only have to address down to the chip level to reach an I/O port in the MCS-4 architecture. But Biewer does not let this space go to waste. He throws the ZERO in on the right-hand side because he needs it as a constant for a later operation. "I kind of fetched two things at the same time," he explained. He believes you are more likely to stumble upon these little opportunities to pack your code efficiently if you do your coding all by hand.

CLC or "clear carry" at line 62 is a necessary precaution. The machine will be looking to see if one of the two side-by-side ONEs being rotated has spilled over into the carry FF, so it is important that the carry start out at zero.

In step 63, the "LD-register-A" instruction loads the motor-drive bit pattern from A into the accumulator where it can be shifted.

The RAL instruction, at 64, shifts or rotates the two ONEs to the left. Since the MCS-4 circuitry places the carry FF in the rotate path, it's possible for the ONEs to get caught in the carry FF and upset the side-by-side pattern that the motor coils want to see. This situation must be detected and corrected.

The JUN instruction at 65, jumps the PC to line 6C where the REVERSE and FORWARD branches of the flow chart meet. There is no need for both branches to duplicate the common steps from now on. The JCN instruction at 6C is used to make the test for overflow into the carry and to direct the program sequence to the correction instruction (at 6E) if necessary. JCN is one of the MCS-4's two decision steps, and so is a proper candidate for implementing the diamond-shaped decision function on the flow diagram.

On the JCN's OPA side the "CO" means that a jump is to be made if the carry FF = 0. Thus, as the arrow out of the left tip of the JCN diamond in the flow diagram indicates, the PC will be jumped around the next overflow correction step to 6F if there is no carry spillover problem. But if there has been a spillover of one of the ONEs into the carry FF, the PC will proceed normally to the next line which contains the ADD instruction that makes the corrective adjustment.

Know your instruction definitions

The ADD instruction, line 6E, directs the contents of register 1 to be added to the accumulator. There is a subtlety here. "You might think that a ONE should be added," Matt said. "I did when I first wrote the program. But if you stop to think about it, you'll realize there will be a ONE already in the carry FF, so you should be adding ONE minus ONE, or ZERO." (That is why the correction in rotate right at line 68 is seven rather than eight.)

Say that your bit pattern in the accumulator was 1100 which, upon shifting left, gave 1000 plus ONE in the carry. If you now add ZERO, you'll get 1001 with the ONE coming from the carry. By the way, the pattern 1001 still has the two ONEs side-by-side as far as the circle of motor coils are concerned.

This new pattern is sent out to the motor by lines 6F and 70. The SRC command at 6F addresses the output port 6. As written in proper MCS-4 coding format, it calls for the pair of index registers, No. 0 and No. 1, to be sent out to the ROM's. But, as we explained awhile ago, you really only want the "6" in register No. 0 to go out, as that is sufficient to select the ROM chip containing port 6. You will recall that the "6" was loaded during line 61. The actual movement of the pattern from the accumulator to output port 6 is effected by the WRR instruction at line 70. There the ONEs excite the motor coils via the drive buffers. The MCS-4's output ports incorporate latches that conveniently hold the pattern until a new one is generated in a subsequent execution of this subroutine, so an external actuator like the motor will be held locked.

But while the output-port latches hold the pattern for the motor, the computer itself will have no way of knowing where it left off in rotating the motor, so the new bit pattern must be saved. This is done with the XCH instruction at line 70. The XCH merely switches the contents of register A and the accumulator. You don't want the old pattern from A in the accumulator, but this XCH happens to be the only command in the MCS-4's repertoire that will move data from the accumulator to CPU index registers.

Your final operation, so far as the movement of the motor is concerned, is to introduce a delay that will allow the motor to settle to its new position before you read the data off the tape.

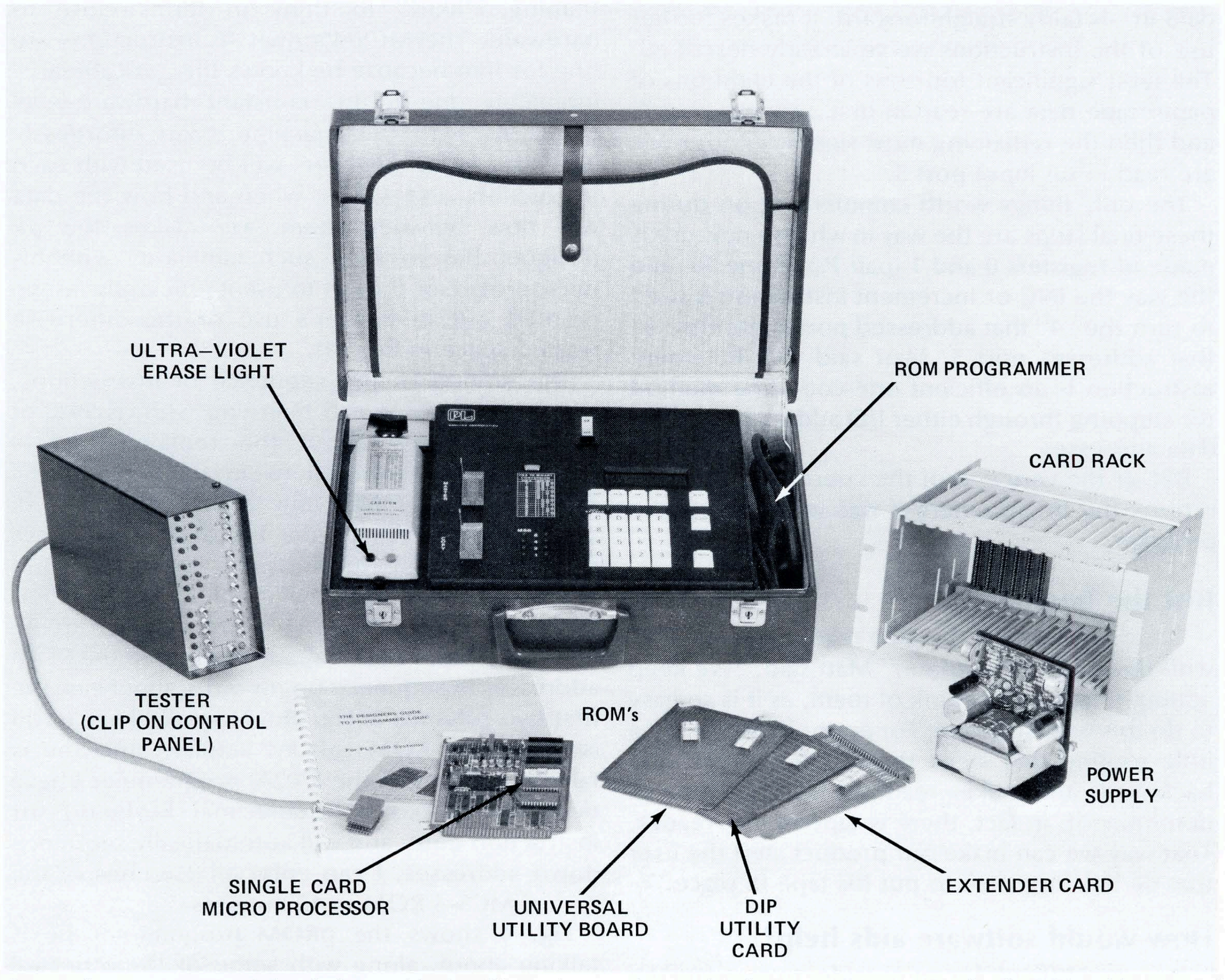


Fig. 4—These are some of the designer aids that Biewer has developed for approaching the MCS-4 in his up-from-hardware manner. The 1702A programmer in the middle is the key tool. It has automatic stepping for the ROM addresses and a hexadecimal keyboard for entering the machine-language instructions.

Matt sets up a register pair (No. P_1) with certain constants, and then jumps the program with a JMS to some ISZ counting instructions.

As the diamond-shaped block in the flow diagram indicates, the ISZ instruction is a decision step—the MCS-4's other one besides the JCN. This "increment-by-one-and-skip-if-zero" instruction is found in one form or other in all computers. Here it directs the CPU to keep looping back and adding ONE to the designated register (2 for the first ISZ) until the content of that register becomes ZERO, then to skip over the loop-back (line 81), and go on to the next step.

Matt must have planned to use this ISZ delay count routine more than once because he shows it as a separate subroutine at a different location (starting at line 80 in our **Fig. 3b**). This represents a subroutine within a subroutine, or two levels of "nesting." As the nesting stack associated with the MCS-4's PC is three levels deep, this presents no problem. The BBL instruction at the end of this short subroutine automatically "pops" the stack and returns the PC to the desired point in the main subroutine—line 76.

The first ISZ can produce up to 345 μsec of delay if register 2's starting count is 0000. The second ISZ can add to this exponentially as it makes the PC keep going back all the way through the first count on each of its loops. Therefore, the total delay, if the second ISZ's register (No. 3) also starts at 0000, can be 5.8 msec. In his actual tape reader application, Matt uses a somewhat more complex arrangement variation of this to produce 10 msec delay.

Reading in the data

The final operation—that of reading the tape data in—is fairly straightforward. It makes further use of the instructions we've already described. The least significant four bits of the eight bits of paper tape data are read in first on input port 4, and then the remaining most significant four bits are read in on input port 5.

The only things worth commenting on during these final steps are the way in which a new use is made of registers 0 and 1 (pair P_0) at line 76, and the way the INC or increment instruction is used to turn the "4" that addressed port 4 into the "5" that addresses port 5. Matt said the increment instruction is an efficient one-code-line method for stepping through either I/O addresses or RAM data addresses.

BBL at the very end of the subroutine (line 7F) returns the PC to where it was in the main line program.

Just the beginning

"Our software for the tape reader doesn't stop with this basic subroutine," Matt said. "We keep adding things as we think of them, as it is so easy to do this with software. For example, we've got a little routine that steps the motor forward and backwards and makes readings from the tape to determine if, in fact, there is tape in the reader. That way we can make our product alert the user that he has forgotten to put his tape in place."

How would software aids help?

It is very difficult for an outsider to evaluate the various software aids. Part of the confusion comes from the way software people seem to imply that these aids help to create or "write" the program. As far as we know, no machine aid can help you select the basic sequence of instructions (like that shown on the right-hand side of **Fig. 3b**), any more than any machine could help the author write this article. But programmers point out that some languages are easier to "think" in. Just like some multi-linguists say they like to think in Italian rather than German, so some programmers feel they can create programs faster when they are thinking in some high-level language like FORTRAN. Once one has gone to some language that is different from what the end user understands (be it a human or a machine), then it is quite sensible to have a computer program that will automatically translate from the easy-to-use language to the final application language.

But Biewer, being a digital circuit designer by training, "likes" to think in terms close to hardware. The MCS-4's own 46 instructions are fine for him because he knows this "vocabulary" intimately due to his constant hardware-level familiarity with this machine. Matt effortlessly visualizes which registers will be used with each instruction, and where, when and how the data will flow between them. He claims the μP designer has to have such familiarity with his microprocessor if he is to use it efficiently (as we pointed out in Biewer's use of the otherwise wasted space in line 61).

The writing of the sequence of instructions, which no machine can help you with, is 99% of the work. How about the remaining 1%—translating the instructions in their mnemonic form into the ONEs and ZEROs that the μP understands? "What's the big deal here?" says Biewer. "I merely run down the instructions and write in the hexadecimal equivalents for their 4-bit code groups (see the third column from the left in **Fig. 3b**). I have already numbered the ROM addresses in sequence on my work sheet (see the first two columns in **Fig. 3b**). Now all I have to do is go to the ROM (pROM actually since he is talking in terms of the 1702A) programmer I have designed. It has a hexadecimal keyboard for instruction entry and will automatically sequence down addresses. I can enter all 256 lines of the typical MCS-4 ROM in 15 minutes."

Fig. 4 shows the pROM programmer he is talking about, along with some of the other μP design and prototyping aids that Biewer has evolved to make his approach easier. □

References

1. *"The Designers Guide to Programmed Logic (For the PLS 400 Systems)," by Matt Biewer, Pro-Log Corp., 852 Airport Rd., Monterey, CA 93940. The textbook that Biewer uses to teach his up-from-hardware approach to programming the MCS-4. It is a very useful supplement to the Intel MCS-4 manual as it provides a second viewpoint that helps the designer see how it can be programmed to perform certain widely-used common functions.*
2. *"Microprocessors are changing your future," EDN, Nov. 5, 1973, pg. 26.*
3. *"Understanding the microprocessor is no trivial task," EDN, Nov. 20, 1973, pg. 42.*
4. *"Understand the 8-bit μP: you'll see a lot of it," EDN, Jan. 20, 1974, pg. 48.*
5. *"Don't overlook the 4-bit μP: they're here and they're cheap," EDN, Feb. 20, 1974, pg. 44.*