EGO: A HOMEBUILT CPU

PART 1: THE SOFTWARE

BY CLIFFORD KELLEY

This homemade CPU sports a clever instruction set

Editor's note: Part 1 of this two-part series covers the instruction set and software aspects of the EGO computer. Next month the author will examine the EGO computer's hardware.

IT IS DIFFICULT TO JUSTIFY building a custom CPU (central processing unit). It won't cost less than an available microcomputer; it will take many hours to design, build, and debug; and, most important, no software will be available for it. However, there are the advantages of being able to incorporate features not found on commercial processors, not to mention the education you would derive from undertaking such a project.

For many years I toyed with the idea of building a CPU. I studied the subject briefly, but reality intruded and I soon forgot it. Then, a few years ago, I took some digital design courses that finally gave me the confidence to determine what my budget could take and to get on with it.

First, I set about defining the requirements. There were only two: Make it simple and make it powerful enough to be useful. These are somewhat contradictory, but in the end I believe EGO fulfilled both.

THE INSTRUCTION SET

The most important part of designing a computer is devising the instruction set. It is the major determinant of the system architecture. As I said, I wanted EGO to be simple yet useful. My definition of useful is the ability to do floating-point operations efficiently. Thus, defining the ALU (arithmetic logic unit) operations was my first step. I wanted the ALU to be able to load, shift, rotate, increment, decrement, add, and subtract at the very least. These, combined with some simple logic operations like AND, OR, and exclusive-OR, gave me a list of 14 instructions. The 74381 4-bit ALU seemed the perfect device. I could use it for the logic and arithmetic instructions. This left me with 16 basic instructions taking up 4 bits of instruction space.

Next came the instruction types. These determine how an instruction defines where the data used by the ALU comes from and where the results will be stored. In order to keep the instruction set simple, I decided to keep all EGO's registers general-purpose. Consequently, there are no restrictions on which registers can be used in what instructions, and the only special-purpose register is register 0, which is used as the program counter. Although I could have constructed an instruction set that loaded data directly to and from memory locations (known on most processors as direct addressing), I felt this would slow down the computer and corner me into including two or three memory-pointing registers (like the HL register pair on the 8080 or the SI and DI registers on the 8088/86) manipulated by a number of special instructions. The only reasonable solution was a large number of general-purpose registers that programmers could dedicate as their needs required. This also forced me to design EGO so that registers could be used for direct addressing (in which the register holds the actual data) as well as indirect addressing (in which the register holds the address of the data).

I decided that EGO would have four different instruc-

(continued)

Clifford Kelley has a B.S. in physics and mathematics from Marquette University. He currently works on missile simulation systems at Rockwell International. He can be contacted at 4127½ Maybank, Lakewood, CA 90712.

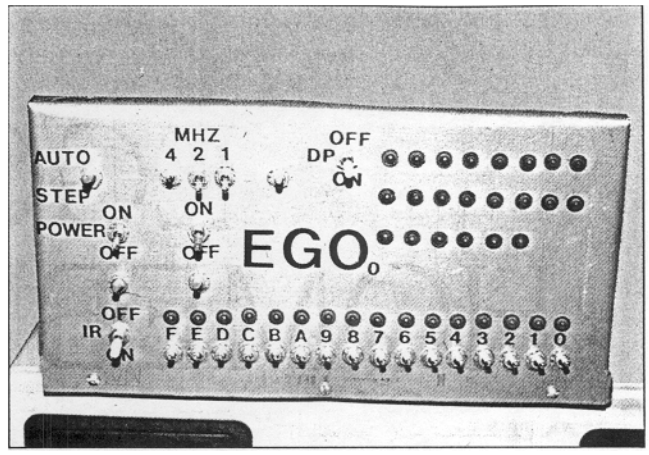


Photo 1: EGO, *the author's homemade* CPU, *sits atop a Tandy* 2000 *that is used for uploading and downloading programs. The switch in the upper left corner allows the author to run* EGO *in single-step mode, and the row of* LEDs *along the bottom can be used to read either* EGO's *data bus or the* CPU's *instruction register.*

tion types: data, register, indirect operand, and indirect result. The data instruction type (often referred to on other processors as the "immediate" instruction type) is composed of two words; the first word holds the instruction, and the second is data. The operation is performed using the data word and the contents of a register, and the result is returned to the register.

All other instruction types require only one word. In the register instruction type, the operands are the contents of two registers, and the result is stored in one of the registers. Indirect operand instructions use the contents of a register as one operand, use the contents of another register as a memory address where the other operand is stored, and leave the result in a register. Finally, the indirect result instruction type functions like the indirect operand type, except that the result is returned to the memory address of the second operand.

The last major decision was how long the instruction word and address lengths would be. Actually, it was an easy decision. To be useful, EGO had to be able to address a large amount of memory; to keep the registers truly general, the instruction length must be the same as the address length. For the address length, I felt the standard 16 bits would be more than adequate. Since EGO would be addressing words, this would give me a total of 64K words, or 128K bytes of memory—twice that available on 8-bit microprocessors. The instruction length is also a multiple of 4, which would make the circuits easier to build and allow 10 bits for register addressing. I felt that a two-address instruction set was a minimum for keeping the instructions general. I would have preferred a three-address instruction set, but that would have left me with only enough bits in the instruction to address eight reg-

(continued)

EGO Instruction Set

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	bit number
: 0 :		0 :		Op Code :		1 :		Condition :			S2 :					Word 1

15															0	bit number
:								Data							:	Word 2

The condition bits refer to the four condition flags that are stored after every ALU operation.

Bit #	Flag	Set condition
8	Z	ALU result is zero
7	N	ALU result is negative
6	P	ALU result is positive
5	C	ALU result overflows (carry

Operation Code	Operation	Mnemonic
0 0 0 0	S2 = Data	LD S2,#Data ****
0 0 0 1	S2 = Data + 1	INC S2,#Data ****
0 0 1 0	S2 = S2 − Data	SUB S2,#Data ****
0 0 1 1	S2 = S2 − Data − $\overline{cy}$	SBC S2,#Data ****
0 1 0 0	S2 = Data − S2	NAD S2,#Data ****
0 1 0 1	S2 = Data − S2 − $\overline{cy}$	NCA S2,#Data ****
0 1 1 0	S2 = Data + S2	ADD S2,#Data ****
0 1 1 1	S2 = Data + S2 + cy	ADC S2,#Data ****
1 0 0 0	S2 = Data xor S2	XOR S2,#Data ****
1 0 0 1	S2 = Data left shift	SL S2,#Data ****
1 0 1 0	S2 = Data or S2	OR S2,#Data ****
1 0 1 1	S2 = Data left rotate	RL S2,#Data ****
1 1 0 0	S2 = Data and S2	AND S2,#Data ****
1 1 0 1	S2 = Data − 1	DEC S2,#Data ****
1 1 1 0	S2 = Data right shift	SR S2,#Data ****
1 1 1 1	S2 = Data right rotate	RR S2,#Data ****

The **** is space for the condition mnemonic code.

Figure 1: *Data instruction types. These are the two-word instructions; all other instructions take only one word.*

isters—not enough registers to allow for both dedicated uses and scratch-pad space for floating-point operations. A two-address instruction set allowed 5 bits to address a register, for a total of 32 registers, which appeared to be more than enough to do the job.

Inevitably, I had to make some compromises in the design of the instruction set. The data instruction type was necessary to load constants into registers. Since I made the program counter part of the general register set and was able to set aside 4 bits in the data type instruction word for condition testing, EGO could perform conditional jumps. This was in keeping with the general ALU structure I had chosen, but some miscellaneous instructions.

(continued)

Condition Code

Condition Bits	Execution Condition	Mnemonic
0 0 0 0	Not carry	I
0 0 0 1	Carry	C
0 0 1 0	Greater than 0	P
0 0 1 1	Greater than 0 or carry	CP
0 1 0 0	Less than 0	N
0 1 0 1	Less than 0 or carry	CN
0 1 1 0	Not 0	PN
0 1 1 1	Not 0 or carry	CPN
1 0 0 0	Equal to 0	Z
1 0 0 1	Equal to 0 or carry	CZ
1 0 1 0	Greater than or equal to 0	PZ
1 0 1 1	Greater than or equal to 0 or carry	CPZ
1 1 0 0	Less than or equal to 0	ZN
1 1 0 1	Less than or equal to 0 or carry	CNZ
1 1 1 0	Unconditional	'blank'
1 1 1 1	Unconditional	'blank'

Figure 2: *Bit settings for condition-flag testing. If the condition is not met, then execution continues with the instruction following the data word.*

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bit number

0 0 0 0 0 0 0 0 Operation Code

Operation Code	Operation	Mnemonic
0 0 0 0 0 0 0 0 0	No operation	NOP
0 0 0 0 0 0 0 0 1	Set carry flag	SCF
0 0 0 0 0 0 0 1 0	Clear carry flag	CCF
0 0 0 0 0 0 0 1 1	Halt	HLT

Figure 3: EGO's *miscellaneous instructions.*

such as set carry, clear carry, halt, and no-operation, are also useful. Fortunately, there was 1 bit left to distinguish between data and miscellaneous instructions.

Thus the instruction set is small but very general. Figures 1 through 5 summarize the instruction set. Bits 15 and 14 hold the instruction type. Bits 13 through 10 determine the operation performed. The two-address instructions (figures 3, 4, and 5) allow one operand from S1 (source 1) and the other from S2 (source 2). It is assumed that the result will be sent to S2. When only one operand is required, S1 is the operand and S2 is the location in which the result is stored. Except for the data instructions, the last 10 bits are divided into two groups: bits 9 through 5 for S1 and bits 4 through 0 for S2.

The first and most complex of the instruction types is

(continued)

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bit number

: 0 : 1 : Op Code : S1 : S2 :

Operation Code	Operation	Mnemonic
0 0 0 0	S2 = S1	LD S2,S1
0 0 0 1	S2 = S1 + 1	INC S2,S1
0 0 1 0	S2 = S2 − S1	SUB S2,S1
0 0 1 1	S2 = S2 − S1 − $\overline{cy}$	SBC S2,S1
0 1 0 0	S2 = S1 − S2	NAD S2,S1
0 1 0 1	S2 = S1 − S2 − $\overline{cy}$	NCA S2,S1
0 1 1 0	S2 = S1 + S2	ADD S2,S1
0 1 1 1	S2 = S1 + S2 + cy	ADC S2,S1
1 0 0 0	S2 = S1 xor S2	XOR S2,S1
1 0 0 1	S2 = S1 left shift	SL S2,S1
1 0 1 0	S2 = S1 or S2	OR S2,S1
1 0 1 1	S2 = S1 left rotate	RL S2,S1
1 1 0 0	S2 = S1 and S2	AND S2,S1
1 1 0 1	S2 = S1 − 1	DEC S2,S1
1 1 1 0	S2 = S1 right shift	SR S2,S1
1 1 1 1	S2 = S1 right rotate	RR S2,S1

Figure 4: EGO's *register instructions.*

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bit number

: 1 : 0 : Op Code : S1 : S2 :

Operation Code	Operation	Mnemonic
0 0 0 0	S2 = (S1)	LD S2,(S1)
0 0 0 1	S2 = (S1) + 1	INC S2,(S1)
0 0 1 0	S2 = S2 − (S1)	SUB S2,(S1)
0 0 1 1	S2 = S2 − (S1) − $\overline{cy}$	SBC S2,(S1)
0 1 0 0	S2 = (S1) − S2	NAD S2,(S1)
0 1 0 1	S2 = (S1) − S2 − $\overline{cy}$	NCA S2,(S1)
0 1 1 0	S2 = (S1) + S2	ADD S2,(S1)
0 1 1 1	S2 = (S1) + S2 + cy	ADC S2,(S1)
1 0 0 0	S2 = (S1) xor S2	XOR S2,(S1)
1 0 0 1	S2 = (S1) left shift	SL S2,(S1)
1 0 1 0	S2 = (S1) or S2	OR S2,(S1)
1 0 1 1	S2 = (S1) left rotate	RL S2,(S1)
1 1 0 0	S2 = (S1) and S2	AND S2,(S1)
1 1 0 1	S2 = (S1) − 1	DEC S2,(S1)
1 1 1 0	S2 = (S1) right shift	SR S2,(S1)
1 1 1 1	S2 = (S1) right rotate	RR S2,(S1)

Figure 5: EGO's *indirect operand instructions.*

data (figure 1). Since only one register is referenced, there is room in the instruction for condition testing. Four condition flags are used to indicate the result of the previous ALU operation: zero, negative, positive, and carry (overflow). These can be tested in any combination, but you will notice that the "no flags" condition is used to test for "not carry" generated (see figure 2). If the condition is not met, the data instruction is ignored and execution con-

(continued)

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	bit number
1 1 Op Code S1 S2	

Operation Code	Operation	Mnen
0 0 0 0	(S2) = S1	LD
0 0 0 1	(S2) = S1 + 1	INC
0 0 1 0	(S2) = (S2) − S1	SUB (S2),S1
0 0 1 1	(S2) = (S2) − S1 − $\overline{cy}$	SBC (S2),S1
0 1 0 0	(S2) = S1 − (S2)	NAD (S2),S1
0 1 0 1	(S2) = S1 − (S2) − $\overline{cy}$	NCA (S2),S1
0 1 1 0	(S2) = S1 + (S2)	ADD (S2),S1
0 1 1 1	(S2) = S1 + (S2) + cy	ADC (S2),S1
1 0 0 0	(S2) = S1 xor (S2)	XOR (S2),S1
1 0 0 1	(S2) = S1 left shift	SL (S2),S1
1 0 1 0	(S2) = S1 or (S2)	OR (S2),S1
1 0 1 1	(S2) = S1 left rotate	RL (S2),S1
1 1 0 0	(S2) = S1 and (S2)	AND (S2),S1
1 1 0 1	(S2) = S1 − 1	DEC (S2),S1
1 1 1 0	(S2) = S1 right shift	SR (S2),S1
1 1 1 1	(S2) = S1 right rotate	RR (S2),S1

Figure 6: EGO's *indirect result instructions.*

tinues. Also, when the data instruction is ignored, the ALU is not used, so the condition flags remain set at their previous values. This makes it easy to test and execute instructions for different conditions.

The miscellaneous instruction set is a subset of the data type. The instruction code is the same as the data instructions, except that bit 9 is zero. These instructions do not fit the general pattern of the rest of the instructions. The no operation, set carry flag, clear carry flag, and halt instructions are defined in figure 3.

The register, indirect operand, and indirect result instructions require only one word. For register instructions, S1 and S2 are direct operands, and the result is stored in register S2. For indirect operand instructions, S1 operand is indirect, S2 is direct, and the result is stored in register S2. For indirect result instructions, S1 operand is direct, S2 is indirect, and the result is sent to S2 indirectly.

The 16 ALU operations are pretty standard. They include the load, increment, subtract, subtract with carry, negate and add, negate and add with carry, add, add with carry, exclusive-OR, shift left, OR, rotate left, AND, decrement, shift right, and rotate right instructions. Although they're very basic, these instructions allow you to perform a number of powerful operations, especially when they're used with the program counter.

The instruction set takes some getting used to. Since I made the program counter a general-purpose register, data instructions can be used as direct or relative jumps. For example, the instruction LD0,#8 loads the program

(continued)

Listing 1: EGO *assembly-language code for direct and relative subroutine calls. Registers* 30 *and* 31 *had to be used to manage a software stack. The instruction* DEC 0,#100 *performs an unconditional jump to memory location* 100.

```
Direct Address Call and Return

Program
.
.
LD 30,0                    Call subroutine program
DEC 0,#100 ------------>100 ADD 30,#3
Address of subroutine      LD (31),30
Rest of program            DEC 31,31
                           DEC 0,(30) ------> Subroutine
.
.
                  Return program
                111 INC 31,31 <------ DEC 0,#111
                    LD 0,(31)

Relative Subroutine Call and Return

Program
.
.
LD 30,0                    Call relative subroutine program
DEC 0,#105 ------------>105 ADD 30,#3
Relative subroutine address LD (31),30
Rest of program            DEC 31,31
                           ADD 30,(30)
                           DEC 0,30 ------> Subroutine
.
.
                  Return program
                111 INC 31,31 <------ DEC 0,#111
                    LD 0,(31)
```

counter with the value 8, which is actually an unconditional jump to memory location 9 (since the program counter is automatically incremented at the end of every instruction). I can use any register as an index register. The simplicity gained by my instruction set resulted in a few strange-looking instructions. I have not yet found a use for some of the data instructions.

Since all of EGO's registers are general-purpose, it becomes the programmer's job to dedicate and keep track of any registers that may be needed for special purposes. Up to this point, I've dedicated only registers 30 and 31 for handling a stack. One of the most important groups of instructions a computer can provide is routine call and return instructions. EGO's flexible structure allows this, although it requires some extra code that most micropro-

continued

Listing 2: EGO *assembly-language code for the interface program.*

Memory Location	Instruction	Comments
0	LD 3,#8188	Load interface address into R3
2	LD 2,(3)	Load instruction into R2
3	AND 2,#255	Use only lower order byte of R2
5	SL 2,2	Multiply R2 by 2
6	SL 1,2	Multiply R2 by 4
7	ADD 1,2	Add for multiply by 6
8	ADD 0,1	Add to PC for mem location
9	NOP	INST #0 NOP loop
10	NOP	
11	NOP	
12	NOP	
13	LD 0,#8	Go to start of NOP loop
15	LD 3,#8189	INST #1 Load address into R3
17	LD 4,(3)	Load data into R4 (address data)
18	LD 0,#8	Go to NOP loop
20	NOP	
21	LD 3,#8189	INST #2 Load address into R3
23	LD 3,(3)	Load data into R3
24	LD (4),3	Load data into (R4)
25	LD 0,#8	Go to NOP loop
27	LD 3,#8189	INST #3 Load address into R3
29	DEC 0,(3)	Go to (data)
30	LD 0,#8	Go to NOP loop
32	NOP	
33	LD 3,#8188	INST #4 Load address into R3
35	LD 2,(4)	Load (R4) into R2
36	LD (3),2	Load R2 into interface
37	LD 0,#8	Go to NOP loop

Memory location 8188 holds a number between 0 and 4. This number vectors execution to routines at locations 9, 15, 21, 27, or 33. The routine at location 9 executes NOP instructions. The routine at location 15 loads register 4 with the contents of register memory location 8189. The routine at location 21 stores the contents of memory location 8189 at the address pointed to by register 4. Location 27's routine transfers control to the program whose start address is stored in memory location 8189. Finally, location 33's routine moves the data at the memory location pointed to by register 4 into location 8188. By storing the proper information at locations 8188 and 8189, another processor connected to EGO's data bus can effectively move programs into EGO for execution and retrieve the results.

cessors execute implicitly. Listing 1 shows the direct and indirect subroutine calls and the return routine.

Programs and Benchmarks

Listings 2 and 3 are examples written in assembly language for EGO. Listing 2 is the source code for the interface program. It resides at the lowest part in memory so that on an interrupt, this program controls what occurs. The interface program allows a simple means of communicating with EGO using memory-mapped I/O (input/output). It recognizes five instructions: no operation, load address data, send data to memory defined by the address data, go to the address data location and run a program, and send data from the address location to the interface program.

Listing 3: EGO *assembly-language code for an 8- by 8-bit multiply.*

Memory Location	Instruction	Comments
2048	LD 5,4	Load first number into R5
2049	LD 0,#8	Return to no-op loop in interface program
2051	LD 6,#8	Load number of bits into R6
2053	LD 8,#0	Clear result register R8
2055 loop	RR 5,5	Rotate rightmost bit into carry flag
2056	ADD 0,#1 I	Jump to shift on no carry
2058	ADD 8,4	Add R4 to R8 to generate the result
2059 shift	SL 4,4	Shift R4 one bit left
2060	DEC 6,6	Decrement the bit count
2061	SUB 0,#8 P	If R6 is positive make another loop
2063	LD 3,#8188	Load address of interface into R3
2065	LD (3),8	Send result to interface
2066	LD 0,#8	Return to no-op loop in interface program

Listing 3 is a simple 8- by 8-bit multiply program controlled by the interface program.

To get an idea of EGO's speed, I wrote a machine-language version of the Sieve of Eratosthenes program. Although EGO doesn't have enough memory to run the entire 16K-byte limit for primes, I ran it to 2K and extrapolated. Based on those calculations, it would take EGO about 3.6 seconds to do 10 iterations up to 16384. This is approximately one-half the time required for a Z80 to make the same calculation running at the same clock speed. With the 4-MHz clock and the average instruction taking between 6 and 9 clock periods, EGO can average about 0.5 million instructions per second.

Since I am interested in simulation and one of the purposes of EGO was to be useful, I wrote some integer and floating-point routines to start an operating system. The floating-point variables have a 32-bit signed mantissa and a 16-bit signed exponent. This provides about nine decimal places, or accuracy with an exponent greater than 9000. Benchmarks on these routines provided these results:

Multiply	0.575 ms
Divide	0.955 ms
Add	0.115 ms

It is interesting to note that while the hardware for EGO took only a few months to complete, the software has taken much longer. Presently, on the Tandy 2000 I have an assembler, an editor, a tiny FORTRAN-type language compiler, and a linker written in BASIC.

Next month I'll discuss the design details of EGO's hardware. We'll take a tour of a machine cycle and see how the hardware supports the EGO instruction set. ■

BY CLIFFORD KELLEY

PART 2: THE HARDWARE

EGO: A HOMEBUILT CPU

How the hardware decodes and executes instructions

Editor's note: This is the second part of a two-part article describing EGO, *the author's homebrew* CPU. *It relies heavily on part 1, published in the September* BYTE, *which describes* EGO's *instruction repertoire in detail.*

The system architecture defines the physical structure of the computer. Figure 1 shows the structure of EGO. The real constraints that the instruction set imposes on EGO are the size of the memory; the number of registers (32); and the width of the data bus, ALU (arithmetic logic unit), and registers.

The IR (instruction) register is used to store the instruction that was read from memory for use by the system controller. The MA (memory address) register stores the address where memory will be accessed. The general-purpose registers are arranged like a small memory bank—only one register at a time can be written to or read from. This factor, along with my desire to keep EGO simple (only one data bus), forced me to use registers A and B as temporary storage for instruction operands, freeing the data bus for transmission of the ALU's operation results either to the TS (temporary storage) register or to a general-purpose register. The TS register is not needed for very fast transfers from the ALU to registers, but it is used as transient storage when data is sent to memory.

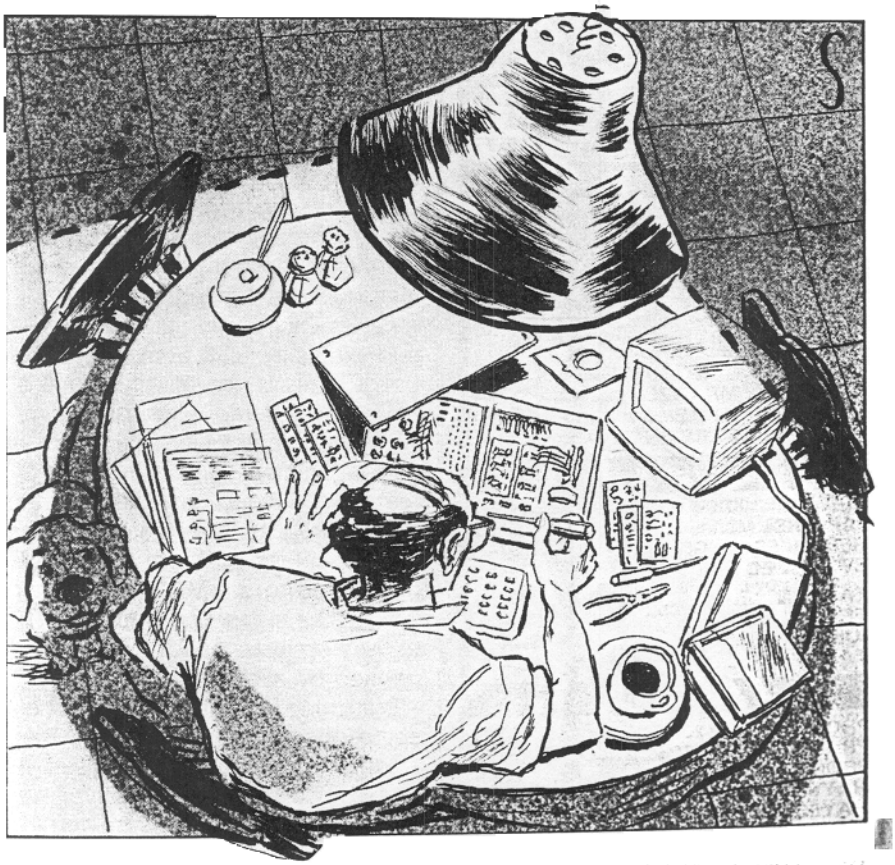
The ST (starting address) register holds the address that EGO accesses when it starts up or is interrupted. Currently it is hard-wired to address 0000 (all addresses are in hexadecimal), but I have made provisions so that it could be loaded by an external device.

EGO SEQUENCER PROGRAM

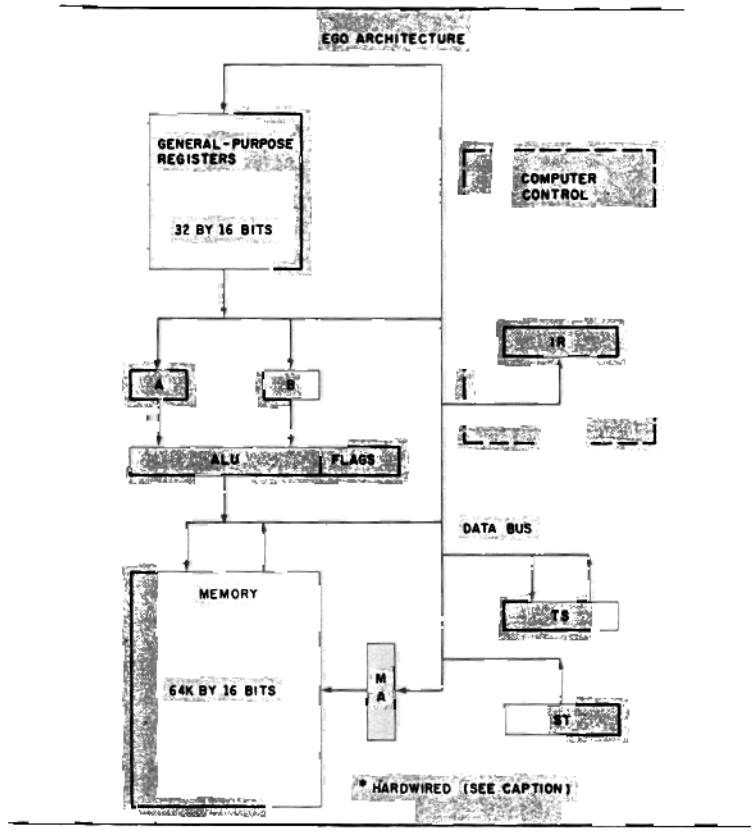
In effect, EGO's sequencer runs a program written in AHPL (a hard-wired programming language; for more information see *Digital Systems: Hardware Organization and Design* by Frederick J. Hill and Gerald R. Peterson, John Wiley & Sons, 1978). The program for EGO is shown in figure 2. It can be implemented either by microprogramming or by hard-wired sequential circuits. For complex machines, a microprogrammed control unit is the only feasible solution. But, since EGO's program has only 22 steps, having

(continued)

Clifford Kelley has a B.S. in physics and mathematics from Marquette University. He currently works on missile simulation systems at Rockwell International. He can be contacted at 4127½ Maybank, Lakewood, CA 90712.



ILLUSTRATED BY STEVE SALERNO



one sequential circuit per control state offers the most direct interpretation of the control sequence, eases maintenance problems, and offers high operating speeds.

As you can see in figure 2, steps 1 and 2 initialize EGO at memory location 0. This is the starting point at power-up, after an interrupt is received, or when EGO restarts after a halt instruction is executed. The ST register is loaded into the program counter (register 0), and in step 2 the contents of the program counter are transferred to the MA register. Step 3 loads the instruction from memory to the IR register and to branches as defined by each instruction type.

Branching from step 3 to step 4 executes the register instructions. Step 4 loads register S1 into register A. Step 5 loads register S2 into register B. Control then transfers to step 17, where the result is computed. This result is sent to the TS register and, since the instruction is not an indirect S2 operation, the result is also sent to register S2. At this point, control transfers to step 20, where the program counter is loaded into register A. Step 21 increments register A and sends the result back to the program counter and into the MA register. Step 21 also checks the interrupt line and, if an interrupt is detected, transfers control to step 1. Otherwise, EGO transfers back to step 3. Since the MA register holds a copy of the program counter, EGO is ready to perform a memory read to fetch the next instruction.

Branching from step 3 to step 6 executes the data instructions. Step 6 loads the program counter into register A. Step 7 increments register A and sends the incremented value

```
Memory  : IR(16);MA(16);TS(16);A(16);B(16);REG(32:16);
          MEM(65536:16);ST(16)
Logic   : ALU(4)
Inputs  : flags,rst,int
Outputs : cy,ldf
Bus     : BUS(16)

 1  BUS=ST,REG(0)<--BUS
 2  BUS=REG(0),MA<--BUS
 3  BUS=MEM(MA),IR<--BUS,-->(IR15IR14IR9,IR15IR14IR9,IR15IR14,
    IR15IR14,IR15IR14)/(19,6,4,11,14)
 4  BUS=REG(S1),A<--BUS
 5  BUS=REG(S2),B<--BUS,-->(17)
 6  BUS=REG(0),A<--BUS
 7  BUS=ALU(INC),REG(0)<--BUS,-->(flags)/20)
 8  BUS=REG(S2),B<--BUS
 9  BUS=REG(0),MA<--BUS
10  BUS=MEM(MA),A<--BUS,-->(17)
11  BUS=REG(S1),MA<--BUS
12  BUS=MEM(MA),A<--BUS
13  BUS=REG(S2),B<--BUS,-->(17)
14  BUS=REG(S1),A<--BUS
15  BUS=REG(S2),MA<--BUS
16  BUS=MEM(MA),B<--BUS
17  BUS=ALU(op code),ldf=1,(REG(S2)<--BUS)/(IR15+IR15IR14),
    TS<--BUS,-->(IR15+IR15IR14)/(20)
18  BUS=TS,MEM(MA)<--BUS,-->(20)
19  cy=IR1IR0,ldf=IR1 ⊕ IR0,-->(IR1IR0)/(22)
20  BUS=REG(0),A<--BUS
21  BUS=ALU(INC),REG(0)<--BUS,MA<--BUS,-->(int,int)/(1,3)
22  -->(rst,rst)/(1,22)
```

Figure 2: EGO's *sequencer program.*

back to the program counter. If the flags logic is false, control transfers to step 20, and the instruction is ignored. (Because the program counter is incremented before the flags are tested in step 7, and because it will be incremented again in steps 20 and 21, the second word of the instruction will be skipped and the program counter will be set properly for the next instruction even if the flags logic is false.) If the flags logic is true, control transfers to step 8. In step 8, register S2 is loaded into register B. Step 9 loads the program counter into the MA register, which now holds the address of the data word of the instruction. Step 10 loads the data word from memory into register A and control transfers to step 17, where the results are computed as I described previously.

Branching from step 3 to step 11 executes the indirect-operand instructions. Step 11 loads register S1 into the MA register. Step 12 loads the memory value, pointed to by the MA register, into register A. Step 13 loads register S2 into register B and transfers control to step 17, where the results are computed.

Branching from step 3 to step 14 executes the indirect-result instructions. Step 14 loads register S1 into register A. Step 15 loads register S2 into the MA register. Step 16 loads the memory value, pointed to by the MA register, into register B. Step 17 computes the result of the operation, but this result is only sent to the TS register. In step 18, the value in the TS register is sent to the memory location pointed to by the MA register. Control then transfers to step 20 and continues as before.

(continued)

The heart of EGO's sequencer is a set of 22 D flip-flops.

Branching from step 3 to step 19 executes the miscellaneous instructions. Step 19 sets or resets the carry flag on a set-carry or clear-carry instruction. For a no-operation instruction, nothing happens and execution continues with step 20. For a halt instruction, control transfers to step 22. Control remains with step 22 until restart is "true." Then control transfers to step 1, where EGO starts instruction execution at location 0.

EGO's Sequencer

Figures 3, 4, and 5 are the logic diagrams of EGO's sequencer. Its heart is a set of 22 D flip-flops, one for every step in the control sequence. The system does not run until the "st" input is true and the "clr" input is false. The "$\overline{on}$" signal is false when any of the flip-flops are set, thus turning off the D input to step 1. This ensures that only one step will be true at any time. The "rst" and the "int" inputs are used to control step 21 and step 22. The outputs from the instruction register are used in a number of places to control the sequence, especially from step 3, where a number of options are decoded. The logic of the flags signal, along with the logic that sets up the address lines for the general-purpose register set, are included in figure 3. Figure 4 shows how the step numbers (the outputs from each flip-flop) are decoded to provide the timing signals that tell EGO what to do. Figure 5 shows the decoding of the op code combined with the sequence steps to provide the signals to control the ALU. Step 17 is the ALU operation step, while step 7 and step 21 are used to set the inputs to the ALU to increment register A.

EGO Registers and ALU

Figure 6 shows the general-purpose register set. Since the 74189 circuit inverts its outputs, the inputs are inverted from the data bus to compensate. The rest of the circuit looks much like a standard RAM.

Figures 7 and 8 show the ALU and registers A and B. One important feature is that register A is hard-wired to the ALU inputs, while the register B outputs are passed through a set of AND gates with the signal "bz" before they are sent to the ALU. I did this so that, by setting "bz" to 0 and performing an OR operation in the ALU, the contents of the A register can be transmitted through the ALU unchanged. This is used for the shift and rotate operations, which are actually performed by an array of tristate

(continued)

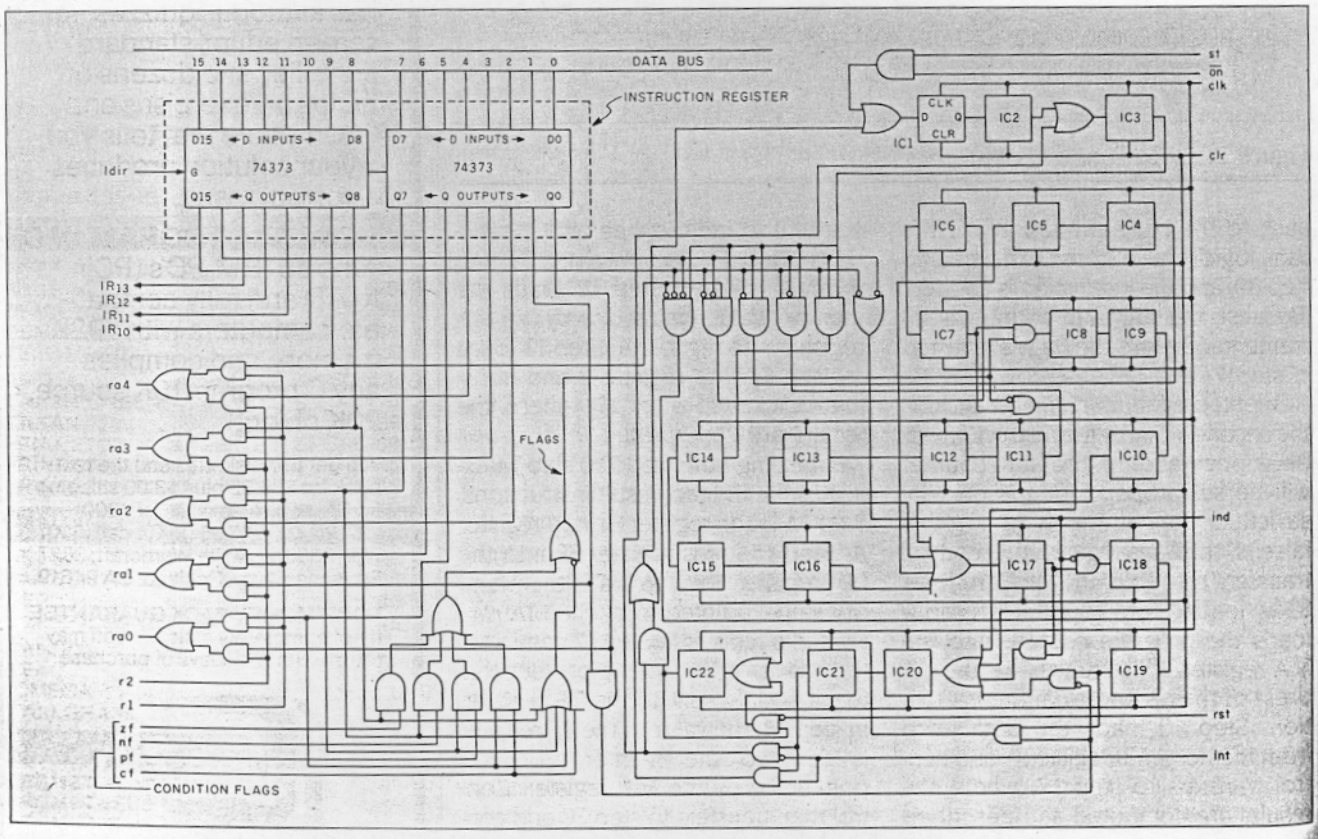
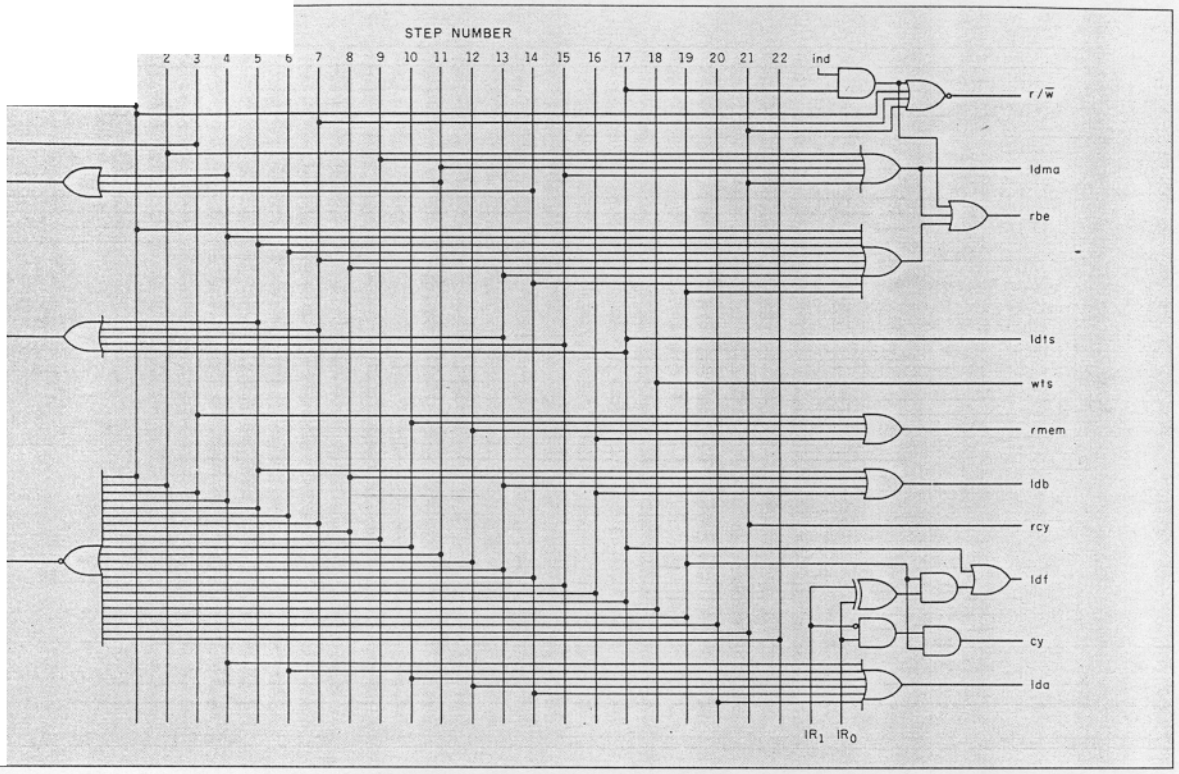
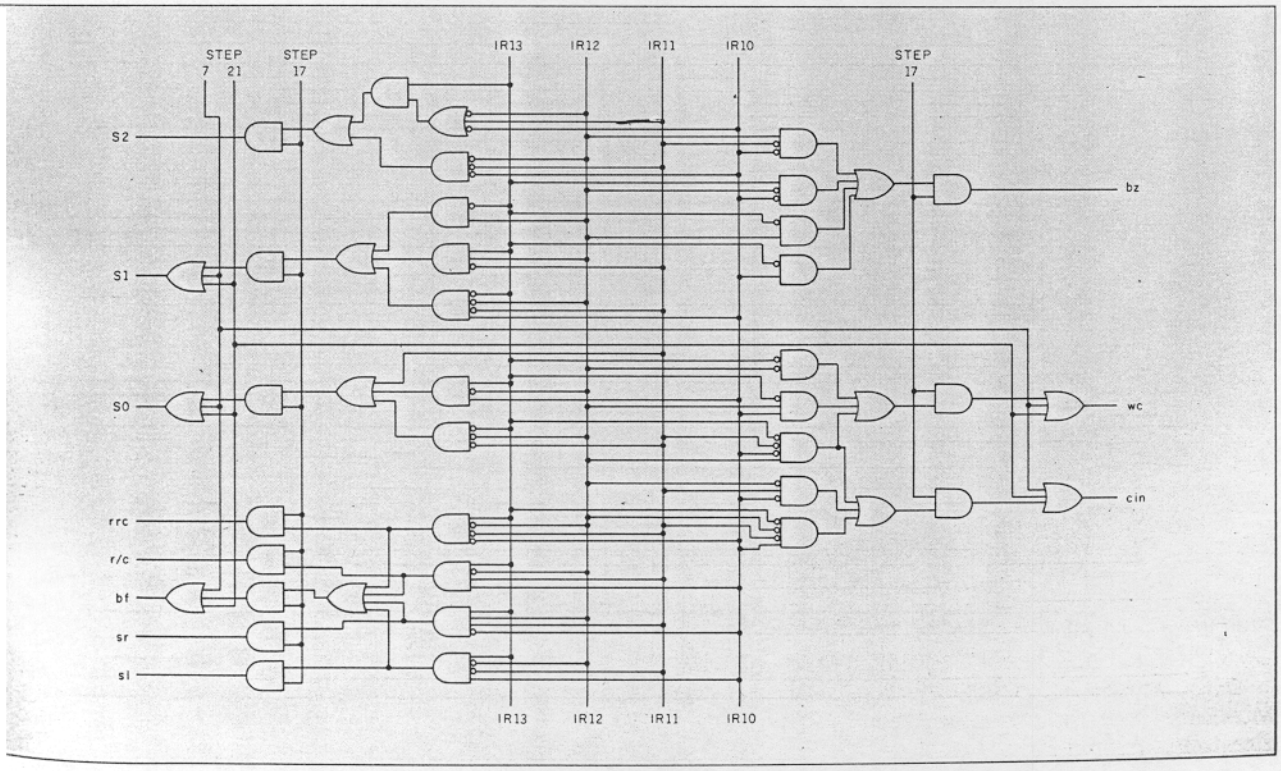


Figure 3: *The main circuit for EGO's sequencer. Note the bank of 22 flip-flops, one for each step in the sequencer program.*



gure 4: *Another portion of the* EGO *sequencer circuit.*



ure 5: *This section of* EGO*'s sequencer performs instruction decoding.*

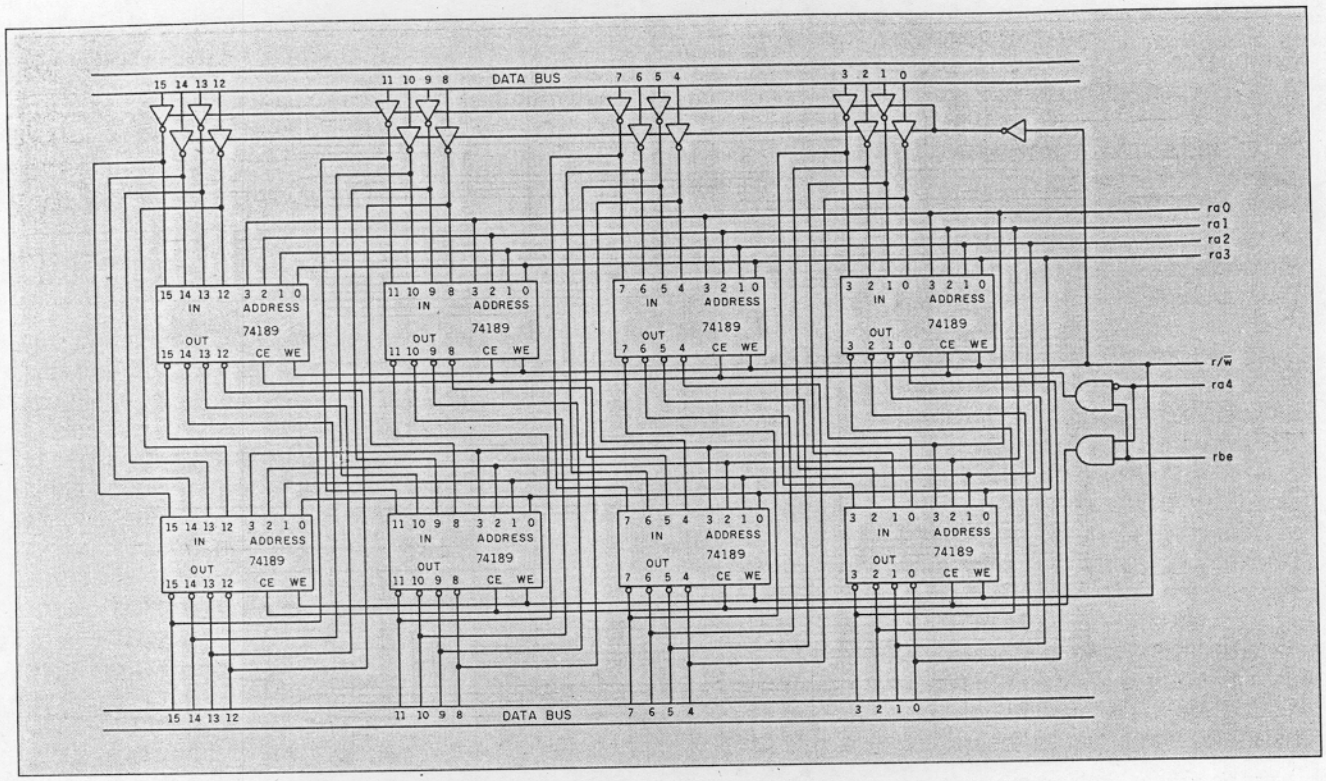


Figure 6: EGO's *general-purpose register set.*

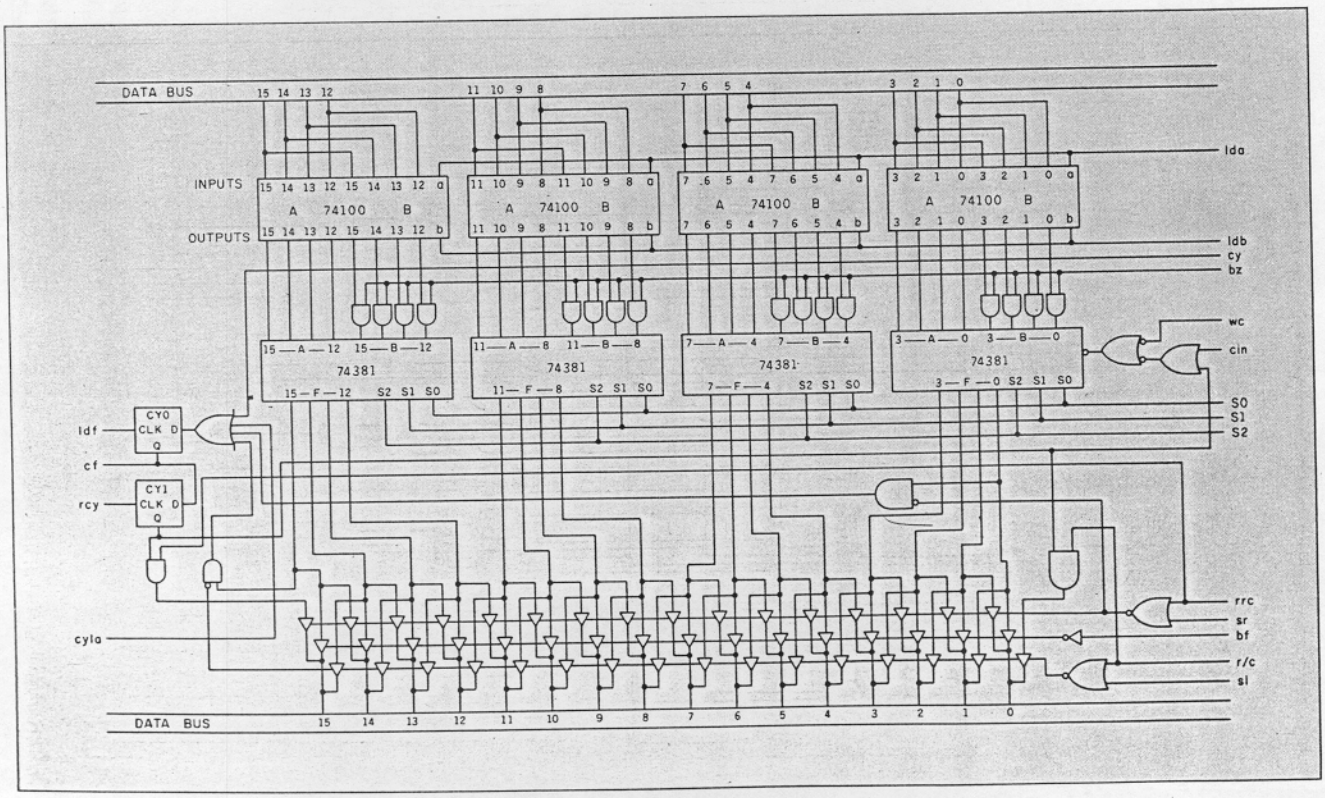


Figure 7: EGO's ALU. *Note the bank of tristate drivers near the bottom used to perform shift and rotate instructions.*

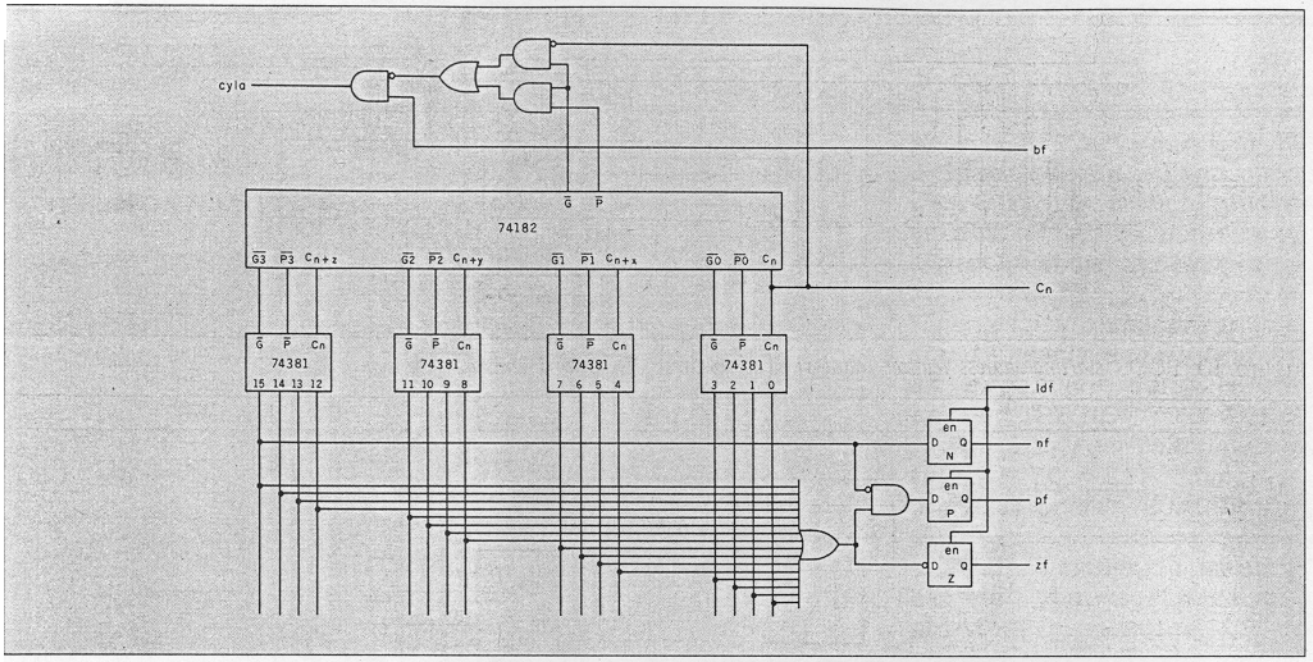


Figure 8: EGO's *circuit for generating carry-look-ahead and condition flags.*

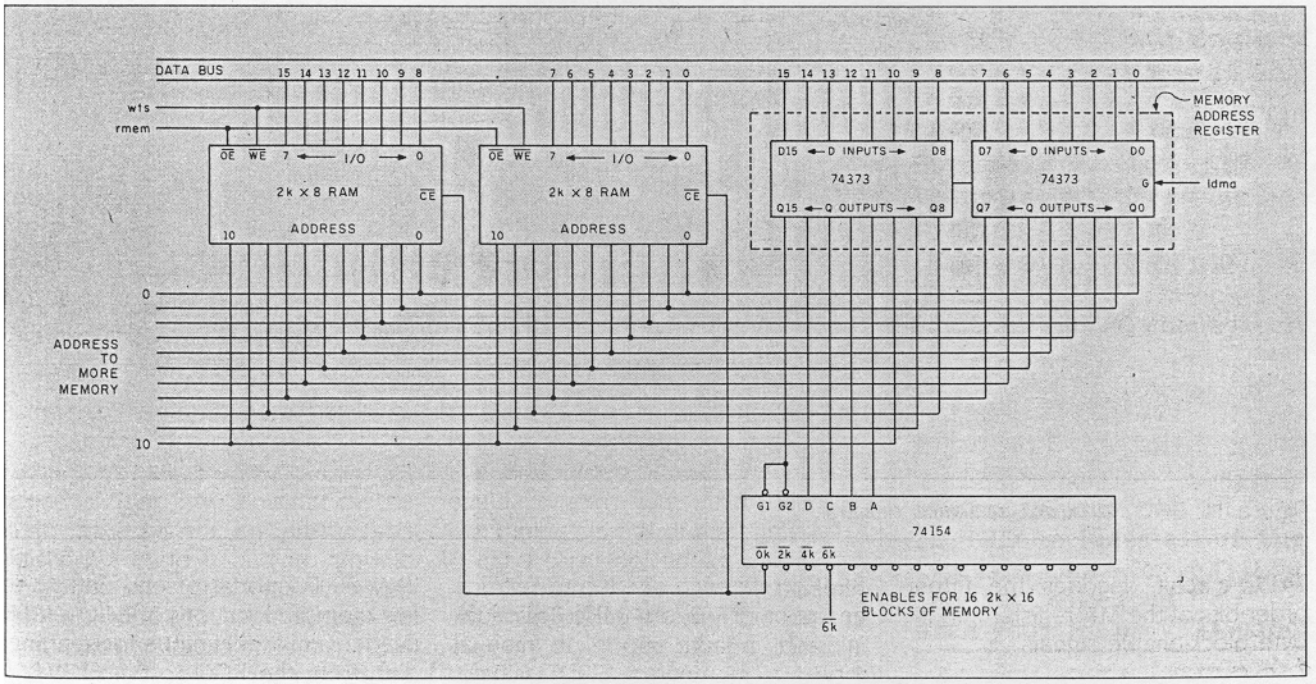


Figure 9: *The circuit for* EGO's *memory and memory-address register.*

drivers attached to the output of the ALU. The carry flag presented special problems because it can be affected by a number of operations (clear carry, set carry, math operations, etc.) Figure 8 shows how the carry-look-ahead circuit is connected along with the other condition flags.

EGO Memory

Figure 9 is a diagram of EGO's memory. The memory-address register can only be written to, and its outputs are always enabled. Currently, EGO is configured around 2K-byte EPROM (erasable programmable read-only memory) and RAM chips. Thus, the

(continued)

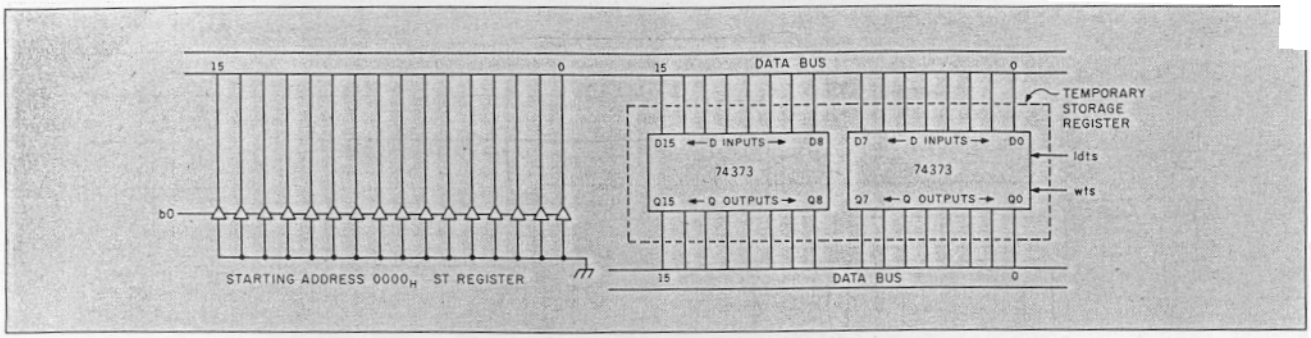


Figure 10: EGO's *starting-address register (hard-wired for address* 0000*) and temporary-storage register.*

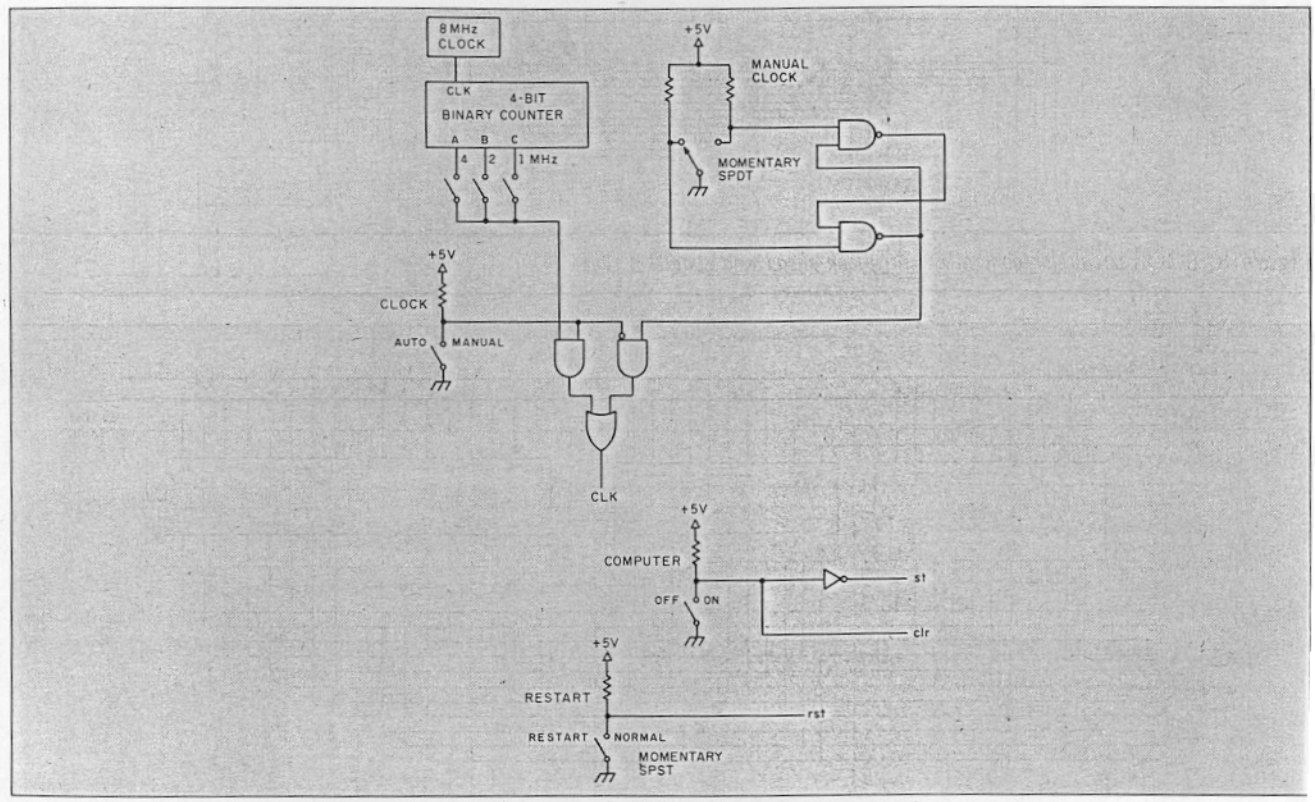


Figure 11: EGO's *clock and front-panel circuitry.*

74154 circuit decodes the upper-order bits of the MA register into 2K-byte block-enable signals.

Miscellaneous Parts

Figure 10 shows how the ST and TS registers are connected to the data bus.

Figure 11 shows the clock and control signals from the front panel. (*Editor's note: Please refer to part 1 of this article for a photograph of* EGO's *front-panel display.*) The clock is derived from an 8-MHz reference divided to provide an option of 1, 2, or 4 MHz. This clock is used in auto mode. In manual mode, a momentary SPDT (single-pole, double-throw) switch on the front panel provides the clock signal. The "st", "clr," and "rst" signals are simple switches.

Figure 12 shows the EGO I/O (input/output) interface. I designed this interface to work with an LNW-80 (a Radio Shack TRS-80 Model I look-alike). I decoded the outputs of the LNW-80 I/O ports to send data to a few memory locations in EGO, and I used a similar technique for sending data from the EGO to the LNW-80. (Although I originally designed EGO to attach to an LNW-80, it currently "talks" to a Tandy Model 2000.)

EGO Present and Future

EGO's memory consists of 2K words of EPROM and 2K words of RAM. I hope to expand this soon, but so far this small amount of memory has no

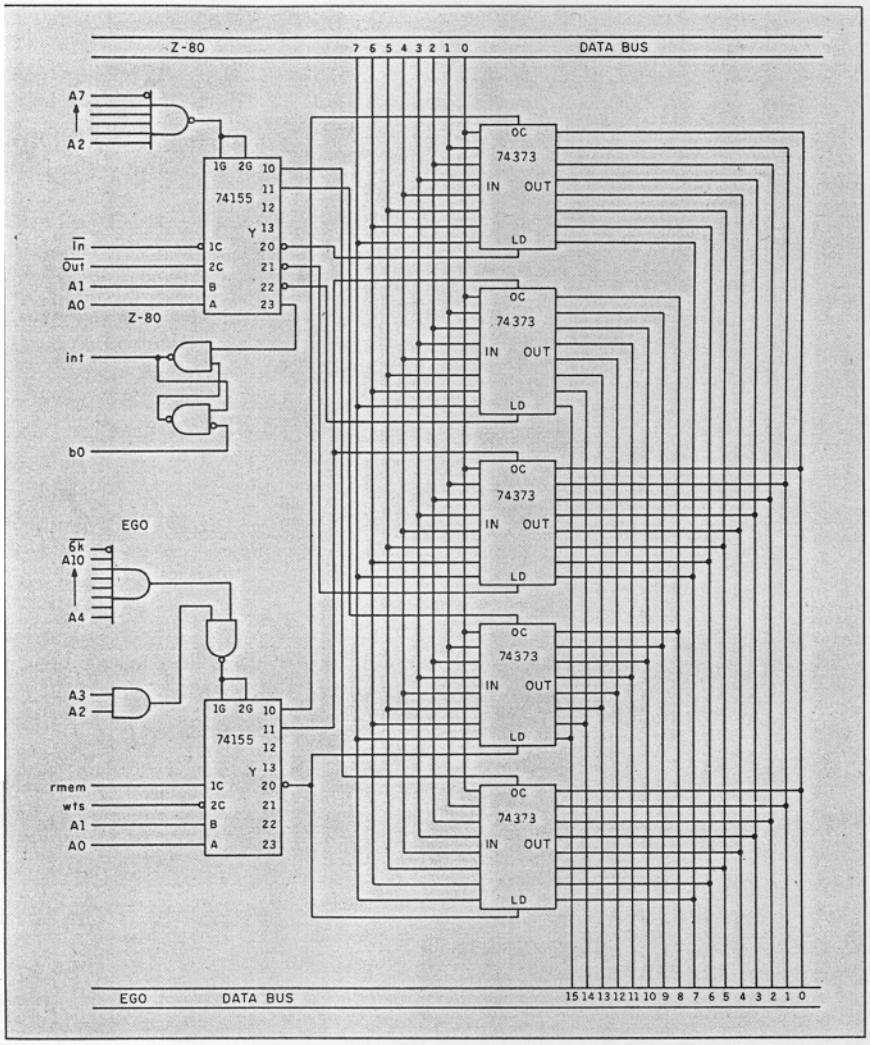


Figure 12: EGO's I/O *interface circuitry.*

hampered my programming even relatively complex programs.

I had originally thought that a clock speed of 1 or 2 MHz would be the limit of the system, but after it was finished I found it had no trouble operating at 4 MHz. This surprised me because the EPROMs have a read cycle of 450 nanoseconds (ns). However, since a valid address from the MA register is available at least 250 ns before the data from memory is requested, no special timing was required to distinguish between EPROM and RAM.

As with any project, I made a few changes along the way, and there are things I would do differently if I were doing it over again. Most of the changes I made were in the sequencer: I eliminated one step entirely and altered a branch. Even now, I could do away with other steps by combining several of them into one. Using a carry-look-ahead circuit, the ALU is probably fast enough to provide a result that could be sent to memory in time for it to be stored. If I were to build another version of EGO, I would incorporate these modifications; but with the system working so well, the incentive to modify it has not been strong. ■

ACKNOWLEDGMENT

I would like to thank Ted Davis, Bill Mealing, and Hutch Hutchison for their technical assistance.