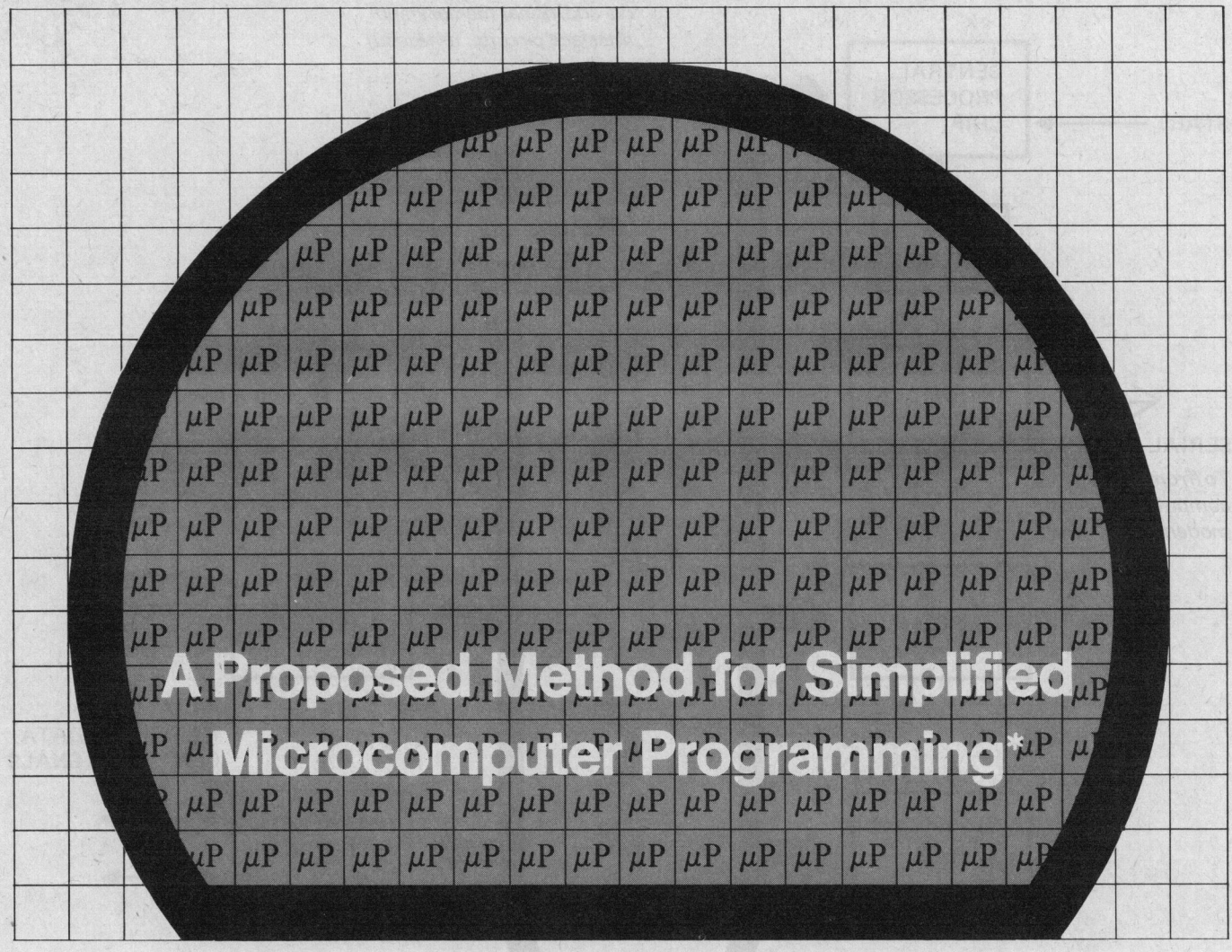


A Proposed Method for Simplified Microcomputer Programming*

Granino A. Korn
University of Arizona

Introduction

A key obstacle to wider application of the new microprocessors is the need to program in assembly language with fairly cumbersome instruction sets. We propose a new technique which will make it easy for applications-oriented non-programmers to generate efficient microcomputer programs without any need to learn assembly language. The user merely specifies an analog-computer-like block diagram whose intuitively meaningful block-operators specify standard mathematical operations (addition, multiplication, integration, etc.) and input/output operations (e.g., analog/digital conversion, switching, sensing). These standard operations are implemented as subroutines in the microcomputer read-only memory (ROM). An interactive editor/translator program running on a small minicomputer translates block-diagram specifications into a simple address table and loads it into the microcomputer memory (RAM, PROM, or ROM). This address table represents the block-diagram operation sequence and specifies successive jumps to the correct standard subroutines in ROM and data fetching/storing operations in the random-access memory.

*A paper based on the material presented here was presented at Wescon 75.

The proposed minicomputer-based programming system is essentially independent of the type of microcomputer used and will also permit convenient and realistic interactive simulation of microcomputer program execution. This will save substantial time and money in checkout and debugging operations.

We believe that the proposed new microcomputer program-development system can significantly enhance usage of microcomputers by applications-oriented non-programmers.

The New Microcomputers

Microcomputers are small digital computers[1] made from a few LSI chips. A central processor may comprise a single LSI chip (microprocessor chip) or several such chips. In addition, a microcomputer system requires some memory for programs and data; clock circuits; and input/output interface selectors, registers, and control circuits (Figure 1). As LSI circuit costs decrease with improved manufacturing yields, microcomputers are replacing much hardwired special-purpose logic in engineering systems. Substitution of computer programming for logic design and design changes, where appropriate, can reduce engineering and production costs. The shortened design cycle is of possibly even greater importance for improved capital utilization and may provide marketing advantages.[2]

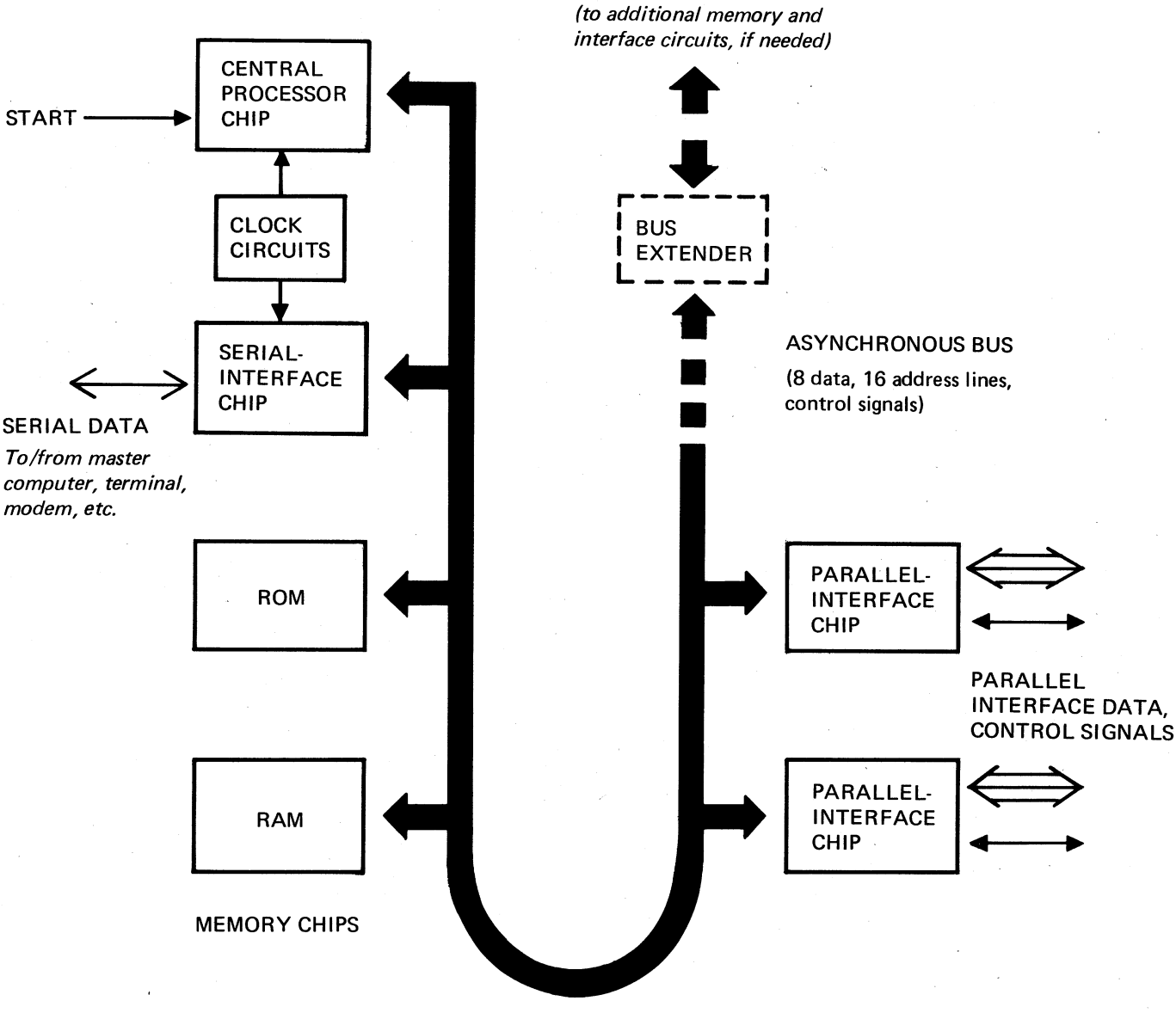


Figure 1. A microcomputer system

The Microcomputer Software Problem

As with all widely applied computers, the cost of producing many thousands of useful applications programs is a very important consideration. Most microcomputers will serve "dedicated" applications–i.e., they will not be used for general-purpose or "end-user" computation. Some dedicated applications may require rare program changes, and their programs will be frozen into safe and relatively inexpensive ROMs. Any other dedicated applications–e.g., instrument and process controllers–will employ microcomputers precisely because of the possibility of reprogramming them for different applications or conditions, or to change a manufacturer's product line in response to competitive products or market conditions. Depending on the situation, new programs may simply require new plug-in ROMs, or one may use electrically programmable ROMs (PROMs), or even random-access memories (RAMs), or a mixture of these.

Unfortunately, microcomputers must usually be programmed in assembly language (or even in machine language), with instruction sets determined more often than not by chip-hardware constraints rather than by programming simplicity. Most frequently, assembly-language programs will be translated by a cross-assembler written in FORTRAN and running on a larger digital computer. One microprocessor manufacturer has also implemented a compiler language (a fixed-point subset of PL-1), but little is known about the efficiency of the resulting microprocessor machine code. This code had better be as efficient as possible, for one would like to optimize execution speed and also minimize the use of memory within the still rather rigid hardware constraints of the small processor. There is, thus, an important requirement for a microcomputer programming system which will permit engineers and scientists to program and reprogram microcomputers without the use of assembly language. Such a programming system ought to satisfy the following requirements:

1. The language should be easy to learn and understand; it should build up complex operations from simple elementary operations well known to engineers and

scientists—i.e., simple mathematical operations and simple I/O operations.

2. The language should be entirely independent of the specific microcomputer used.
3. The programming system should generate microprocessor code as efficient in the use of time and memory as that generated by a good assembly-language programmer.
4. From point of view of cost and program safety, it would be good to program as many standard elementary procedures as possible in ROM.

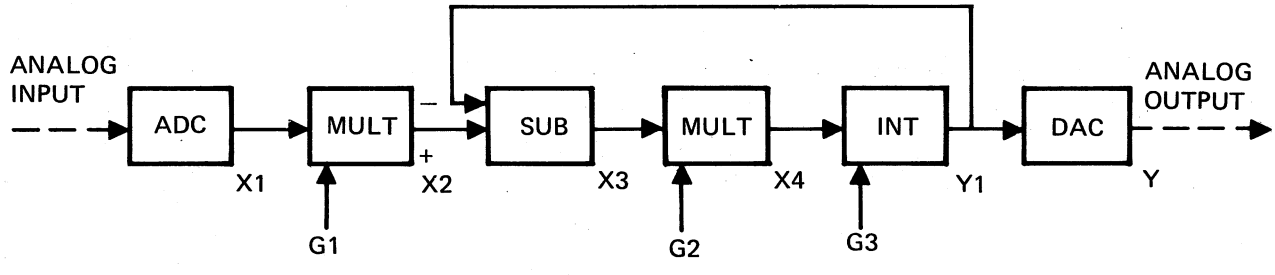
This paper proposes new microcomputer programming systems meeting the above requirements. In order to begin with a nicely defined problem area, we propose these programming systems for microcomputer instrumentation controllers intended to schedule, control, and process measurements in physics, chemistry, biology, or electrical engineering. Actually, the new programming systems apply equally well to computer process control. They are, in fact, not restricted to microcomputer programming but will also generate efficient and fast executing programs for minicomputers. Much of the proposed programming system is, indeed, based on our experience with efficient minicomputer block-diagram programming systems,[3] and the proposed research will contribute in this area as well.

connected to produce samples of a low-pass filtered output quantity

$$Y1 = G1 \frac{1}{\frac{1}{G2\,G3}\frac{d}{dt} + 1} X1$$

which is then reconverted into an analog output Y(T) by a digital-to-analog converter (DAC). Such a block diagram represents a complex operation in terms of simpler elementary block operators. This type of model representation is familiar and acceptable to many engineers and scientists.

For digital computer programming, the block diagram of Figure 2a is represented in Figure 2b by an ordered list of all the block operators together with their input and output variables, outputs first. Figure 2b then represents our program in a block diagram language. Note that the block operators are in "procedural order" with the state-variable-producing integrator last. Some block-diagram programming systems, such as DARE II and DARE/ELEVEN,[4,5] will automatically sort block statements into procedural order—i.e., so that each block input appearing in the program has been properly computed by a block operator appearing above it. However, this can also be done by hand, if desired.



(B)

DAC	DUMMY, Y1
ADC	X1
MULT	X2, X1, G1
SUB	X3, X2, Y1
MULT	X4, X3, G2
INT	Y1, X4, G3

Figure 2. Block diagram (a) and sorted block-diagram-language program (b) for a simple digitally-implemented filter for analog voltages. The program involves input/output as well as mathematical operations. Note that the MULT operator occurs twice, with different arguments and results.

A Block-Diagram Programming System

Figure 2 illustrates the proposed programming technique by a simple, concrete example. Figure 2a is an analog-computer-like block diagram which describes the action of a simple microprocessor acting as a primitive digital filter with analog input and analog output. An analog-to-digital converter (ADC) produces a digital sample X1 of its analog input X(T) every DT seconds, an I/O operation which will automatically synchronize the digital computation with real time. Next, multiplier (MULT), subtractor (SUB), and integrator (INT) block operators are

Note that the variables X1, X2, X3, Y1 in our block diagram vary with the time variable T and must be updated every DT seconds. The complete sequence of block-operations of Figure 2 is, therefore, repeated every DT seconds (possibly twice or more per DT interval if we employ, say, Runge-Kutta integration[4,5]). A block-diagram programming system, then, includes "canned" routines INIT and RUN which will start the program at the time T = 0, repeat it every DT seconds, and then terminate operation at a time $T = T_{MAX}$. INIT comprises a run-time data-input routine, which permits us to supply the program with the required parameters G1, G2, G3, with the initial value of the

integrator output Y, and with the parameters DT and T_{MAX}.

To translate the block-diagram program of Figure 2 for execution by a digital computer, the programming system must supply the canned routine RUN, which repeatedly calls the sequence ("derivative file") of Figure 2b. Each block operator in the list could be considered as an *assembly-language macro*[1] which generates corresponding assembly-language statements like

```
LOAD ACCUMULATOR WITH
 CONTENTS OF              X2
SUBTRACT CONTENTS OF      Y1   } SUB X3, X2, Y1
STORE RESULT IN           X3
SKIP IF ADC CONVERSION
 COMPLETE
JUMP BACK ONE STEP             } ADC X1
READ ADC
STORE RESULT IN           X1
```

This is, indeed, a good method for minicomputers with plenty of memory and without user microprogramming.[4,5] To save memory when block operators (like MULT in Figure 2) recur, we can represent each operator by a subroutine, or by a subroutine in control memory (microprogram).[1,2] In either case, subroutines for standard block operators can be implemented in read-only memory for efficient, high speed storage, and program safety. For machines capable of fast microprogram execution, representation of block operators by microprograms is especially efficient,[2] since it will save references to slower main memories. Even where standard operator subroutines are stored in ROM, it is still possible to add special-purpose subroutines in random-access memory.

Note now that the block-diagram language program in Figure 2 neither involves assembly-language programming nor does it refer to a specific computer implementation. Standard block-operator subroutines or microprograms for a given computer would be written once and for all by a system programmer. They could then be sold in various combinations on ROM chips; it is only necessary to access them and to provide them with inputs and outputs as efficiently as possible. This will be the job of our language translating system.

An Efficient Technique for Calling Block-Operator Subroutines

To generate a subroutine sequence representing the block diagram of Figure 2, our translator might produce macros which generate conventional subroutine calls within calling sequences,[1] thus

```
JUMP TO SUBROUTINE        INIT
G1
G2
G3
YINIT
JUMP TO SUBROUTINE        RUN
DT
TMAX
JUMP TO SUBROUTINE        DAC
Y1
JUMP TO SUBROUTINE        ADC
X1
JUMP TO SUBROUTINE        MULT
X1
G1
X2
....
....
```

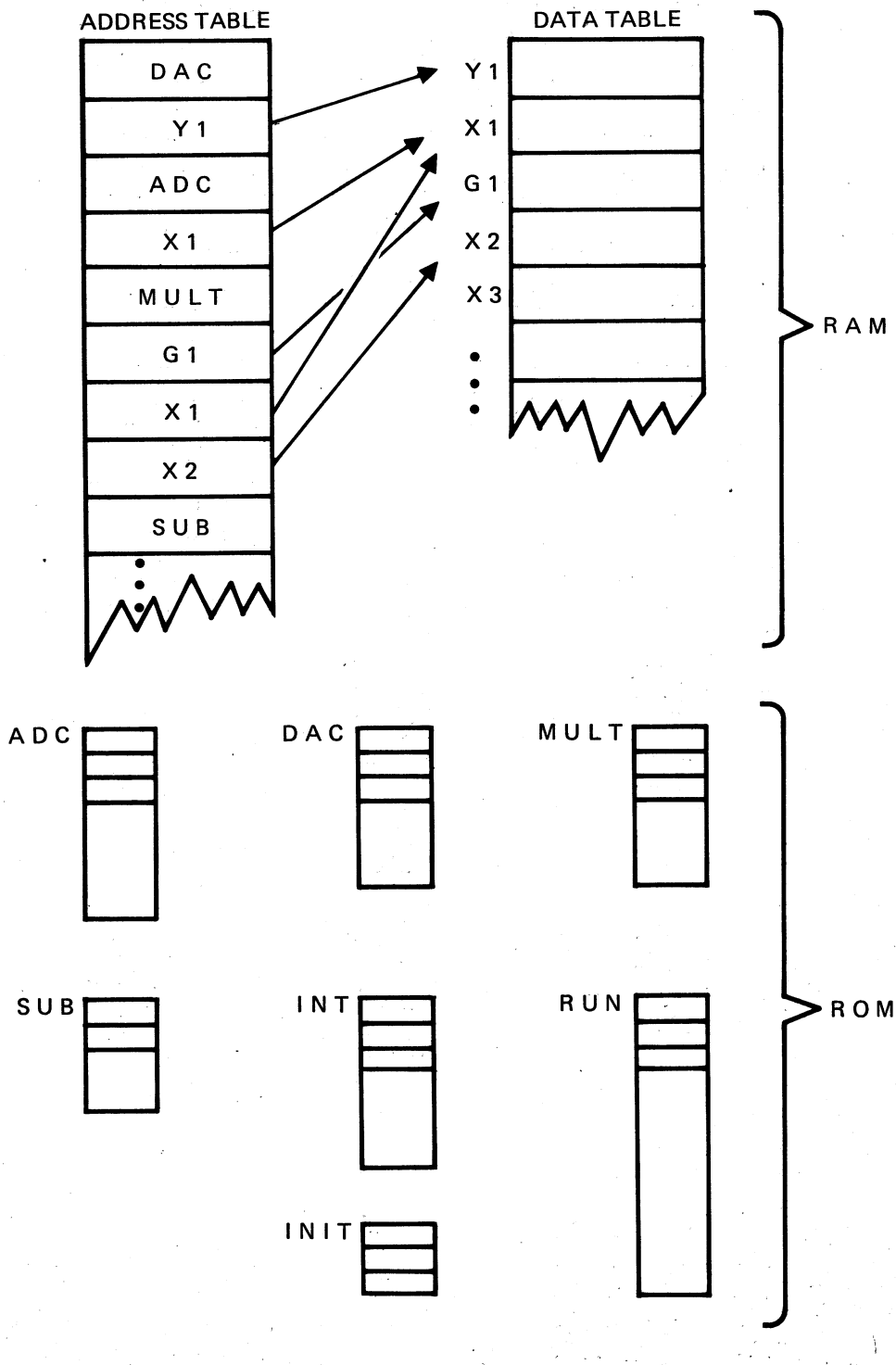
where G1, G2, . . . are addresses of data locations in RAM.

But the block-diagram programming technique of Figure 3 (suggested by University of Arizona graduate student S. Conley in the course of a study of minicomputer block-diagram programs) is more elegant and especially well suited to microcomputer implementations.

Referring to Figure 3, we can see that the ordered block-operator macros of Figure 2 now generate not subroutine calls and calling sequences, but simply a table of the addresses where the appropriate subroutines and data are kept. The entire program, then, is represented by an address table loaded into the microcomputer RAM. Here the addresses ADC, SUB, ... are addresses of our canned, standard subroutines, while addresses like Y1, X1, X2, . . . are addresses pointing to a data table prepared in RAM by the translator. This table contains block-operator inputs and outputs generated during each cycle through the block diagram or input from terminals, digiswitches, or dials prior to computation. Output data may also be placed into the data table, which can be regarded as a COMMON area for the subroutines (since each data item may be referred to more than once).

To access subroutines and data, we initially load an index register with the starting address of our address table. Each subroutine, referring again to Figure 3, fetches successive data items with the operation

```
LOAD ACCUMULATOR, INDIRECT VIA INDEX
 REGISTER
INCREMENT INDEX REGISTER
```



which may involve a single instruction (PDP-11-like machines) or several instructions. Data are stored with the similarly pre-indexed indirect STORE operation

STORE ACCUMULATOR, INDIRECT VIA INDEX REGISTER

INCREMENT INDEX REGISTER

To access succeeding routines, each subroutine ends with

JUMP, INDIRECT VIA INDEX REGISTER

INCREMENT INDEX REGISTER

Figure 3. Complete program section to run the filter of Figure 2 in terms of "canned" subroutines DAC, ADC, MULT, SUB, and INT in the read-only memory of a microcomputer. RUN is another canned ROM routine, which initializes and then repeats the entire sequence between T = 0 and T = T_{MAX}. The specific program for the filter is a simple address table containing successive jump and data addresses in RAM; also in RAM there is a table for all variable data and parameters needed by the program.

Each subroutine ends with a pre-indexed indirect jump to the next subroutine. Data are fetched and stored with pre-indexed, indirect fetches and stores, and the index register increments after each operation. Thus, indeed, indirect jumps and data moving operations step through the correctly ordered address table to implement and repeat the program.

Note that only address and data table need be entered into the RAM of our microcomputer, with all standard, pure-procedure block-operator subroutines and the initialization and RUN subroutines stored permanently in ROM. To enter a new program, a new address table will be loaded into the microcomputer, probably most often through a serial I/O interface (available on a single chip for many microcomputers, Figure 1). The required address table program may come from a minicomputer, from a keyboard terminal, from cassette or paper-tape storage, or even from a larger digital computer through a serial communication link. For a dedicated microcomputer, the address table can also be stored in ROM.

To initialize an experiment or instrument operation, the INIT subroutine can read initial data into their proper data-table locations from their typed or stored input or from interface registers interrogated by this routine—e.g., from dials or digiswitches. Computer operation is then started with the first access to the RUN subroutine, either manually or by an interrupt. The correct references to the data table (Y1, X1, G1, X2, ...) in Figure 3 are also generated by the translator.

Minicomputer-Based Programming Systems

We propose microprocessor block-diagram programming systems running on a small minicomputer (16-24K 16-bit words, disk, cassette-tape or floppy-disk operating system) with a simple alphanumeric CRT terminal for program entry and editing. Such a programming system comprises

1. an editor program, which permits interactive creation and editing of block-diagram programs by typing on the CRT screen, with optional hardcopy output;
2. a file-handling system, (really part of the editor) which permits storage and retrieval of named programs or parts of programs;
3. a block-diagram translator, which translates block-operator statements into the required microcomputer address and data tables;
4. a microcomputer loader, which loads the translated object program (address and data tables) into the microcomputer RAM (for more firmly dedicted operation, the address table is loaded into programmable ROMs (PROMs) through a PROM-loader unit);
5. a run-time system for simulated execution of microcomputer programs on the minicomputer, with appropriate CRT or hardcopy output (see below).

Our concepts of block-diagram programming and translation are closely related to our earlier NSF-supported studies of interactive simulation and instrumentation-control systems for minicomputers. In particular, DARE/ELEVEN (Ref. 5 and Appendix A) is a new minicomputer simulation/instrumentation system which incorporates all the above-listed program features with the exception of a microcomputer loader. As shown in Appendix A, DARE/ELEVEN is more elaborate than we need for our microcomputer-programming application. But DARE/ELEVEN is carefully constructed from independent subprogram modules and could be readily cut down to size, while a suitable microcomputer loader is added. Application of DARE/ELEVEN would produce useful results very quickly.

MICRODARE, A New Simplified Block Diagram Programming System

MICRODARE, still in the paper-study stage in the course of an earlier NSF study grant, is a much simpler block-diagram programming system, which seems especially well suited for the microcomputer programming application. While DARE/ELEVEN implements its editing, translation, loading, and run-time phases as successive overlays under a disk operating system, *MICRODARE is developed from a conventional EXTENDED BASIC system* which, as is well known,[1] permits simple interactive editing and file manipulation, as well as computation and I/O. Additions to the BASIC system perform the block-diagram-translation and microcomputer-loading operations. Fast minicomputer simulation of microcomputer block-diagram program execution is also possible. MICRODARE, less elaborate than DARE/ELEVEN, will not require multiple microcomputer core overlays, so that no elaborate disk operating system is needed.

Machine-Independent Simulation of Microprocessor Operation for Program Development and Checkout

The larger microprocessor manufacturers have developed simulation programs which simulate microprocessor program execution on a large (usually timeshared) digital computer. The big machine is fed the microprocessor assembly-language program and then simulates its execution, producing suitable error diagnostics. Input/output operations are not easily simulated in this manner, and the elaborate simulation program must be written all over again for each new microprocessor.

By contrast, once the standard block operators are available in ROM or RAM, all microprocessors look exactly alike to our minicomputer programming system. Specifically, every microprocessor looks like a standard block-diagram operation. Also, the minicomputer block-diagram programming system can readily execute the block-diagram program (including suitable I/O operations) itself and at the same time provide convenient oscilloscope or recorder output of block-operator-output time histories, indicate overloads, and display error diagnostics. (In fact, such simulation/instrumentation operations were the original purpose of the block-diagram systems developed in our laboratory.)

Our minicomputer programming system will thus provide a neat, machine-independent technique for interactive development and checkout of microprocessor programs written in terms of specified block operations.

Minicomputer block-diagram programming systems like DARE/ELEVEN and MICRODARE also make it quite easy to develop and test new types of block operations.[3,5] The new routines will then have to be written for the

microprocessor, typically using a cross-assembler. But this work would ordinarily be done by a system programmer rather than by applications-oriented microprocessor users.

Proposed Approach

Since the software translation system is the same for any type of microprocessor, it will be economically possible to try the new programming systems with several different types of microprocessors. We suggest starting with second-generation 8-bit processors like Intel 8080 or Motorola 6800, both of which have very convenient I/O chips and byte-organized memories. We should like to implement—as a radically different and very interesting system—block operator subroutines as microprograms using the four-bit-slice architecture and writable control store of a National Semiconductor IMP-16L which will permit experimentation with a complete microprocessor version of a MICRODARE system.[2] Finally, Western Digital Corporation PDP-11-emulating microprocessors would be eminently well suited to instrumentation control because of their flexible instruction set.[1]

The programming system will require a minimal size PDP-11/40 processor to translate block-diagram programs; at least one microprocessor cross-assembler (Motorola 6800) also runs on PDP-11.

Possible Follow-on Programs and Related Studies

A number of small but potentially very fruitful follow-on studies involve the development of block operators for special instrumentation applications. We should be particularly interested in on-line Fourier analysis, amplitude-distribution analysis, and the computation of correlation functions and spectra, with the associated display operations. CAMAC (standardized instrument interface) crate controllers for parallel or serial data transfers will be an especially useful and cost-saving microprocessor applicaton.

A simple translator refinement is a provision for killing data-fetch operations made redundant by the fact that a preceding block-operator has left the desired block output in the accumulator. This can save substantial execution time, because most microprocessors fetch 16-bit data words in two separate 8-bit bytes.

The block-diagram operations discussed here are all fixed-point operations, ordinarily using a two's-complement fraction code to represent analog quantities. Floating-point operations are perfectly possible, but would require more time and computer memory.

The most interesting follow-on work would involve multiple-microprocessor systems. The minicomputer initially employed for inexpensive program translation could serve as a control processor for a system of several microprocessors operating in data acquisition and process control, all using the original block-diagram programming methods. ■

Granino Arthur Korn has an international reputation in the area of computer systems based both on his 27 years as a researcher and professor at the University of Arizona and as author of numerous widely used text books. These include *Computer Handbook* (Korn & Korn), *Digital Computer Users Handbook* and *Minicomputers for Engineers and Scientists* (McGraw-Hill, 1973). His recent research includes developing parallel processing systems for simulation and instrumentation.

References

1. Korn, G. A., *Minicomputers for Engineers and Scientists*, McGraw-Hill, N. Y., 1973.

2. Collins, D. C., E. Garen, and G. A. Korn, *Seminar Notes on Minicomputers and Microcomputers*, Technology Service Corp., Santa Monica, Calif., 1974.

3. Korn, G. A., "Ultra-fast Mini-computation with a Simple Microprogrammed Block-diagram Language," *COED/ASEE Transactions*, March, 1974.

4. Aus, H. M., and G. A. Korn, "The Future of Continuous-system Simulation," *Proc. AFIPS/FJCC*, 1969.

5. Martinez, R., DARE/ELEVEN, Ph.D. Dissertation, Electrical Engineering Dept., The University of Arizona, 1974.

6. Halling, H., "CAMAC Serial System with Programmable Crate Controller," *Proc. CAMAC Symposium*, Commission des Communautés Européenes, 1974.

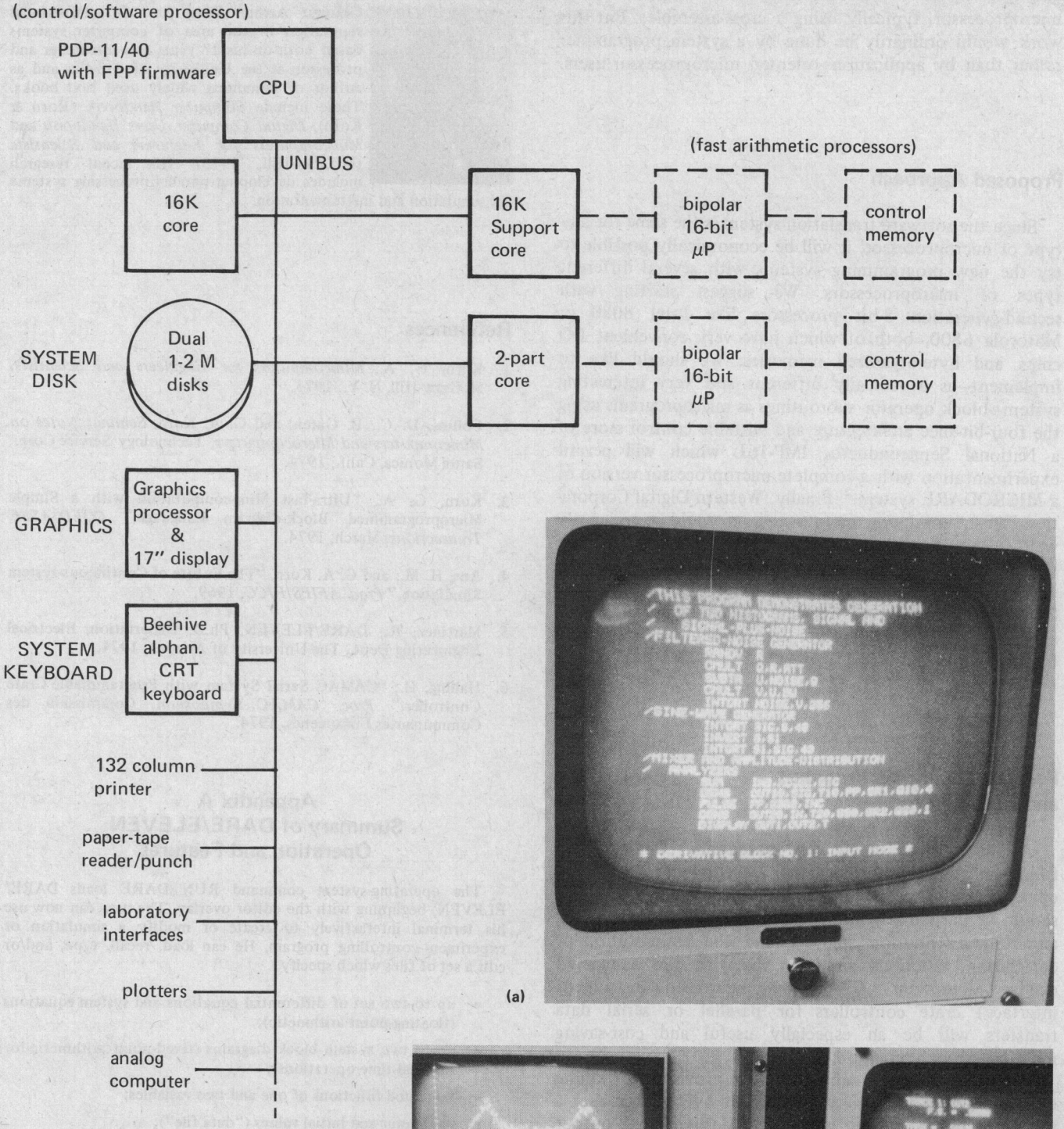
Appendix A
Summary of DARE/ELEVEN Operation and Features

The operating-system command RUN DARE loads DARE/ELEVEN, beginning with the editor overlay. The user can now use his terminal interactively to create or modify a simulation or experiment-controlling program. He can load, recall, type, and/or edit a set of files which specify

- up to two set of differential equations and system equations (floating-point arithmetic);
- up to two system block diagrams (fixed-point arithmetic for fast real-time operations);
- tabulated functions of one and two variables;
- parameter and initial values ("data file");
- a FORTRAN program for controlling multiple run studies;
- a simulation-output program (listings, displays, plots, saving time histories for comparison with later runs);
- FORTRAN subroutines and assembly-language subroutines and macros for appropriate use by experienced programmers.

A user's stored files will be linked to his sign-on user number and user-group number for protection. Files for a complete problem will be entered into the active file list (AFL) associated with a problem identification code (PIC). Such a problem specification might involve only a set of differential equations and a data file for a simple application, or it might include any combination of the above files. Since DARE systems will always default to the simplest possible situation, the more sophisticated language features need not be seen or known by users interested only in simple applications.

On the other hand, experienced programmers can easily create new block-diagram operations, including real-time operations on



(a)

(b)

Figure A-1. Proposed research installation

Figure A-2. Program entry (a), and probability-density display (b) for a DARE II Monte-Carlo simulation

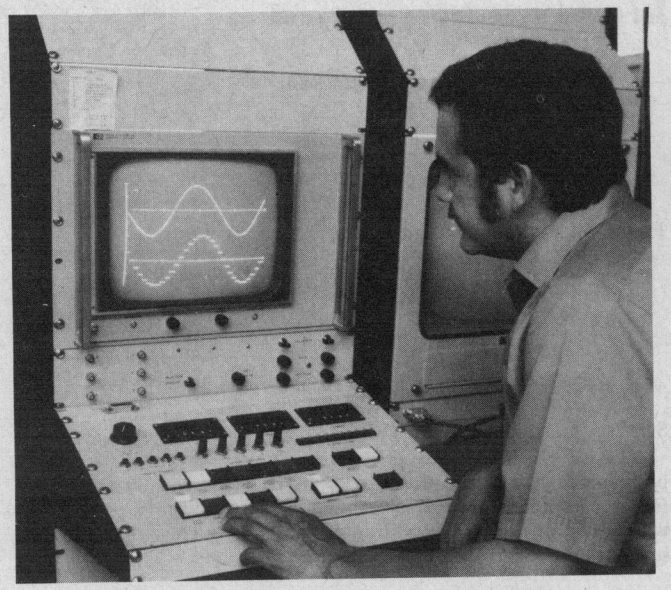


Figure A-3. DARE study of a sampled-data system

external equipment, as assembly-language macros and can even modify the DARE system itself by entering appropriate program segments under DARE subprogram names. This will not disturb these DARE system programs permanently; for permanent changes, DARE system files must be accessed through the disk operating system, using a special user number. DARE/ELEVEN is written in terms of modular subroutines, so that substitution of new subprograms is easy.

The keyboard command CO(MPILE) will, in several overlays, translate (precompile) the active files of a problem into FORTRAN and/or assembly language, compile and/or assemble, link programs and library routines, and load the resulting object program as a part of the DARE/ELEVEN run-time system.

Switch options entered with the COMPILE command produce translator output, listings, and load maps. Block-diagram system users also have a choice between faster assembly and somewhat slower execution, and slower (optimized) assembly and faster execution.

In this overlay, the user can still display or print the data file (parameters and initial values), display, enter, and change parameters and initial values, call or suppress disk storage of solution time histories, and change integration methods without recompilation.

A simulation run, or a FORTRAN-controlled sequence of runs (simulation study), is now obtained through the keyboard command RUN. There will be a run-time display if one was programmed; more elaborate output can be programmed or be obtained in response to keyboard commands in the output overlay.

Note that parameters, initial conditions, *and* integration methods can be changed without recompilation before a new RUN command is given.

Appendix B
Pre-indexed, Indirect Operations for Motorola 6800

Introduction This appendix shows program segments implementing the pre-indexed, indirect operations required for the block-diagram programming technique described above. Unfortunately, the 6800 has no indirect addressing. Therefore, 5 to 8 bytes and 18 to 23 cycles for eacy 16-bit load, store, etc., operation are the price of the pure-procedure code used in our ROM subroutines. A machine with true pre-indexed, indirect operations

would, of course, do much better. The PDP-11, for instance, needs only single 1-word instructions:

```
JMP   @ (R1)+        ; jump
MOV   @ (R1)+,R0     ; load
MOV     R0, @ (R1)+  ; store
```

A National Semiconductor PACE or IMP would also be more convenient, especially since the latter permits not only indirect addressing and 16-bit operations, but also microprogrammed multiplication.

Note also that, unlike those in the PDP-11, our 6800 programs increment the pointer *before* each operation.

Address Pointer in Memory Define byte locations IX, IX+1 on page 0 as a *pointer*. IX contains the most-significant byte of the 16-bit address. Note that IX and IX+1 are "direct," one-byte addresses.

			Bytes	*Cycles*
LDX	IX	; get pointer	2	4
INX		; increment pointer	1	4
INX		; again	1	4
STX	IX	; store it	2	5
LDX	0,X	; indirect address	2	6
			8	23

We can then do a *pre-indexed, indirect jump:*

JMP	0,X	2	4
	total	10	27

This can be replaced by JSR SUBR in the main program together with RTS in each subroutine for greater efficiency.

But pre-indexed, indirect load, and store will be more efficient than subroutine calling sequences, since these would still require pre-indexed, indirect operations, plus transfers between stack pointer and index register. To load a 16-bit word into accumulators A and B, we use

LDAA	0,X	; most significant	2	5
LDAB	1,X		2	5
			12	33

and to store accumulators A and B, we use

			Bytes	*Cycles*
STAA	0,X	; most significant	2	6
STAB	1,X		2	6
			12	35

For 16-bit addition into accumulators A and B, we use

ADDB	0,X	; least significant	2	5
ADCA	1,X		2	5
		Total	12	33

Subtraction is similar; using SUBB and SBCA.

Use of Stack Pointer as Address Pointer If the 6800 stack pointer is not used for subroutine or interrupt return-address storage (and this is indeed the case in many of our block-diagram programs), we can employ it as the address-table pointer. Thus the indexing sequence is replaced by

INS		; increment pointer	1	4
INS		; again	1	4
TSX		; transfer to index	1	4
LDX	0,X	; indirect address	2	6
			5	18

In this case, bytes and cycles needed are reduced to

	Bytes	*Cycles*
Jump	7	22
Load	9	28
Store	9	30
Add, Subtract	0	28

Note that it will now be more necessary to employ the pre-indexed, indirect jump rather than JSR/RTS, since the stack pointer is no longer available.

Complete Macros As an example, a complete macro for adding two 16-bit numbers from memory into accumulators A and B, and storing the result in memory, would combine the above program segments as follows:

(a) Pointer in Memory			*Bytes*	*Cycles*
	JSR	SUM	3	9
SUM	LOAD		12	33
	ADD		12	33
	STORE		12	35
	RTS		1	5
			40	115
(b) Using Stack Pointer as Address Pointer				
SUM	LOAD		12	33
	ADD		12	33
	STORE		12	35
	JUMP		10	27
			46	128

It appears that the pointer-in-memory method is preferable in this case, and in any routine involving not more than four memory-reference operations such as load, store, add, etc. Note finally, that the overhead due to the pre-indexed, indirect operations will be less serious in routines such as multiplication, which also involve only two operands but a more complicated operation.