AD-A067 824 NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA F/G 9/2
DIRECT EXECUTION LANGUAGE ARCHITECTURE FOR SIMULATION.(U)
JAN 79 R MARTINEZ
UNCLASSIFIED NOSC/TR-367 NL

END
DATE FILMED
6-79
DDC

LEVEL II

(12)

# NOSC

NOSC TR 367

NOSC TR 367

**Technical Report 367**

# DIRECT EXECUTION LANGUAGE ARCHITECTURE FOR SIMULATION

R. Martinez

15 January 1979

Interim Report: October 1977 — September 1978

Prepared for
Chief of Naval Material



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

RR GAVAZZI, CAPT, USN
Commander

HL BLOOD
Technical Director

## ADMINISTRATIVE INFORMATION

The work presented in this report was performed from October 1977 to September 1978. This study was funded under NOSC IR project number ZR64.

Released by
B. A. Bologna, Head
Weapons Technology Division

Under authority of
D. A. Kunz, Head
Fleet Engineering Department

## ACKNOWLEDGEMENTS

The author wishes to gratefully acknowledge the continued support of Drs. Eugene P. Cooper and Ellen E. Kuhns, Code 013, at the Naval Ocean Systems Center. Thanks go to Division Heads, Mr. Jim Gilbreath and Dr. Douglas Chabries for their support. The author acknowledges the design work which was performed by Ron M. Hidinger (Code 6353), Jack M. Zyphur (Code 9131), and Mark J. Perrin (Code 6353). A special thanks is due to Dr. Granino A. Korn, University of Arizona, for his assistance with the interpreter/compiler software.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER: NOSC Technical Report 367 (TR 367) | 2. GOVT ACCESSION NO. |
| 3. RECIPIENT'S CATALOG NUMBER: rept. | |
| 4. TITLE (and Subtitle): Direct Execution Language Architecture for Simulation | 5. TYPE OF REPORT & PERIOD COVERED: Interim Oct 1977 - Sep 1978 |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s): R. Martinez | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Ocean Systems Center, San Diego, CA 92152 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; ZR00001; ZR0140802/ZR64 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Material Command, Washington, D.C. 20362 | 12. REPORT DATE: 15 Jan 1979 |
| | 13. NUMBER OF PAGES: 30 |
| 14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: 34p. | 15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The study presented in this report addresses the application of direct execution computer architecture concepts using low-cost, large-scale-integration (LSI) to continuous system simulation. Basic architecture schemes for direct execution systems were investigated and applied to a demonstration interactive simulation language. A dual processor direct execution system was designed and built using an LSI-11 microcomputer, a 16K X 16 dual port memory, a microprogrammed AMD 2900 bit-slice direct execution processor, and a 16 X 16 multiplier chip. The system features a distinct separation of interpretive/compiler software and simulation run execution firmware linked via the dual port. The system is demonstrated via several simulation examples including realtime control, optimization, and torpedo dynamics problems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

393 159

## SUMMARY

### OBJECTIVE

In recent years high level languages for continuous system simulation have been designed to execute on general purpose digital computers which have traditional von Neumann-type architectures. This approach has produced varying results including a multiple layer of system software required to implement and execute the simulation language. In most cases, users have been constrained to perform noninteractive simulations at a high cost. In an attempt to alleviate some of these problems, this study addressed the application of direct execution computer architecture concepts using low-cost, large-scale-integration (LSI) to continuous system simulation. This report summarizes work performed up to October 1978.

### RESULTS

Basic architecture schemes for direct execution systems were investigated and applied to a demonstration interactive simulation language. A dual processor direct execution system was designed and built using an LSI-11 microcomputer, a 16 K × 16 dual port memory, a microprogrammed AMD 2900 bit-slice direct execution processor, and a 16 × 16 multiplier chip. The system features a distinct separation of interpretive/compiler software and simulation run execution firmware linked via the dual port. The system is demonstrated via several simulation examples including realtime control, optimization, and torpedo dynamics problems.

### RECOMMENDATIONS

The study revealed several key points in applying direct execution architectures to continuous system simulation. A direct execution system, similar to the one demonstrated, can be used to replace analog computer-type functions in hybrid simulation systems, process control and instrumentation, and simulation model development systems. Extensions to the architecture can be made from miltiprocessor simulation systems. Scale-free floating point direct execution system can be microprogrammed using the same hardware configuration.

## CONTENTS



## ILLUSTRATIONS

## INTRODUCTION

Traditionally, high level languages (HLL) have been implemented on von Neumann architecture computers which consist of a central processing unit, input/output bus, user interfaces, and program/data memory. In particular, high level simulation languages for the solution of dynamic continuous systems have been implemented on large mainframe and minicomputer systems in batch and interactive modes (1, 2, 3). Simulation programs written in high level languages have been first translated into compiler or assembler-based languages, relocatable code produced, and linked with a simulation run-time library in order to create a run module executable by the host processor. Figure 1 depicts this process where the host processor also executes the editor, translator, compiler, assembler, linking loader, and input/output operations. This report summarizes a different approach to the design and use of high level simulation languages and represents the application of large-scale-integration and advanced computer technology to continuous system simulation. The results depict a digital system design which can directly execute a high level simulation language.

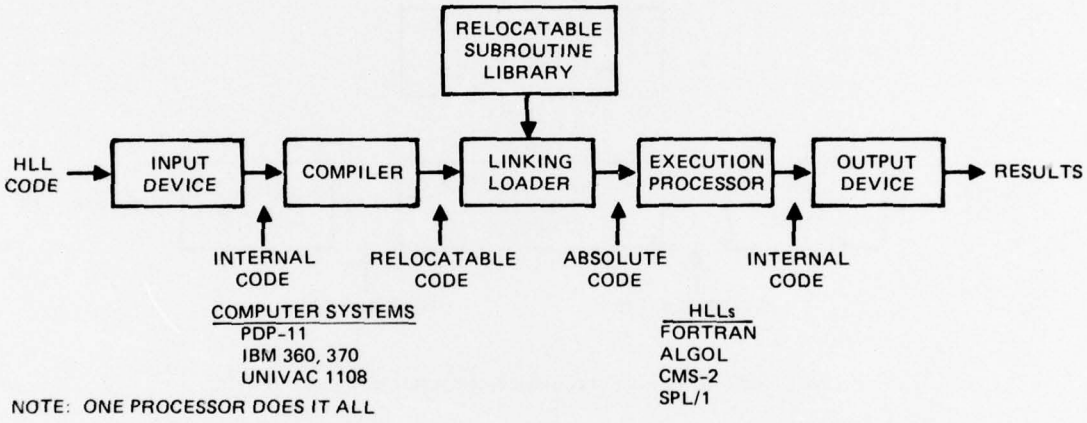



Figure 1. HLL in a traditional von Neumann architecture.

Several large-scale-integration technology characteristics contribute to the application of direct execution to simulation. First, bipolar technology parts offer a high speed computational capability. Microprogrammable bit-slice microprocessor parts can be used to design a computer architecture optimized for continuous system simulation. Second, large-scale-integration special purpose function units are commercially available which can be used to compute highly repetitive operations and mathematical functions, such as multiply and sine/cos, which frequently arise in simulation. A typical example is the 16-bit multiplier chip from TRW (4). In addition, support systems for large-scale-integration devices are available which enable the user to spend added time on system development rather than on development tools.

Direct execution computer architectures represent a departure from the von Neumann influenced relationship between computer and high level language. In a direct execution computer architecture, the high level language is the machine language of the system (5). The high level language constructs are directly executed in hardware without the need for separate software parts such as compilers, assemblers, and linking loaders. Figure 2 depicts this process. The language constructs may be executed on a symbol-by-symbol (or token-by-token), statement-by-statement, or procedural section basis. The basic functional elements which are used to perform direct execution of a high level language include (6):

- input/output processor
- text editor
- lexical scan and syntax processor
- control processor
- arithmetic processor
- data structure processor
- processor intercommunication buses and memories
- special function computation units

Large-scale-integration makes implementation of these functional parts feasible. For example, interpreters of the 1960's and early 1970's required large memories and were slow in execution. However, today's advanced memory technology features 64K bit chips and sub-microsecond cycle times (7).

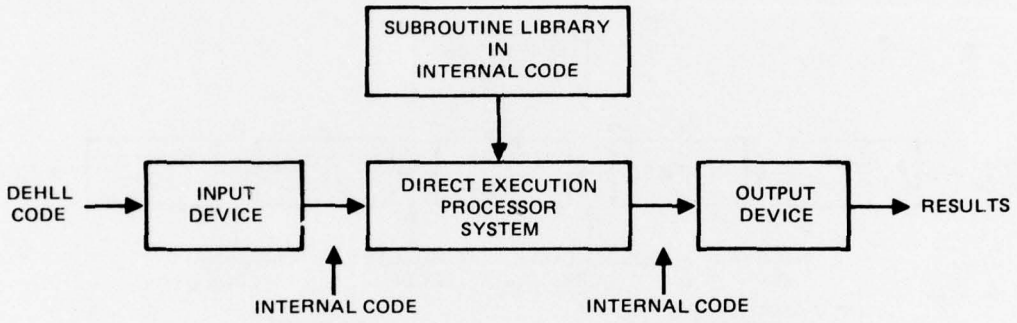



Figure 2. HLL in a direct execution architecture computer system.

The content of this report expands on the points described above including a brief overview of previous direct execution computer architecture work. The reasons for the application of direct execution computer architecture concepts to simulation are discussed. A specific high level simulation language is presented and used as a basis for the detailed LSI design of a dual processor direct execution system. Finally, potential application areas for direct execution computer architecture are explored and follow-on work proposed.

## BRIEF HISTORY OF DIRECT EXECUTION COMPUTER ARCHITECTURE

Useful in this area as background material, the text edited by Chu (8) contains a comprehensive survey of the field. A reference list compiled by Carlson (9) summarizes the history of direct execution computer architectures and their relationship to high level programming languages. A few samples of work performed are briefly reviewed here.

Research in the relationship of high level programming languages to computer architectures started as early as the 1950's, however, the first computer system design with the concept of direct execution computer architecture was the Burroughs B5500 in 1961. The B5500 used the concept of a hardware stack to store executable statements expressed in reverse Polish notation and served as a predecessor for many designs. An ALGOL 60 design described by Anderson (10) in 1961 was an extension of the B5500 architecture and consisted of three stack memories and pointers. The three stacks were used to process control

states, arithmetic operators, and operands. The operator stack was used to resolve operator precedence as arithmetic statements were interpreted. Another detailed ALGOL 60 design was formulated by Chu in 1973 (11). The functional parts of Chu's ALGOL computer are shown in Figure 3. The partitioning of the architecture into functional elements performing specific operations and control has led to Chu's current implementation approach using large-scale-integration (6).

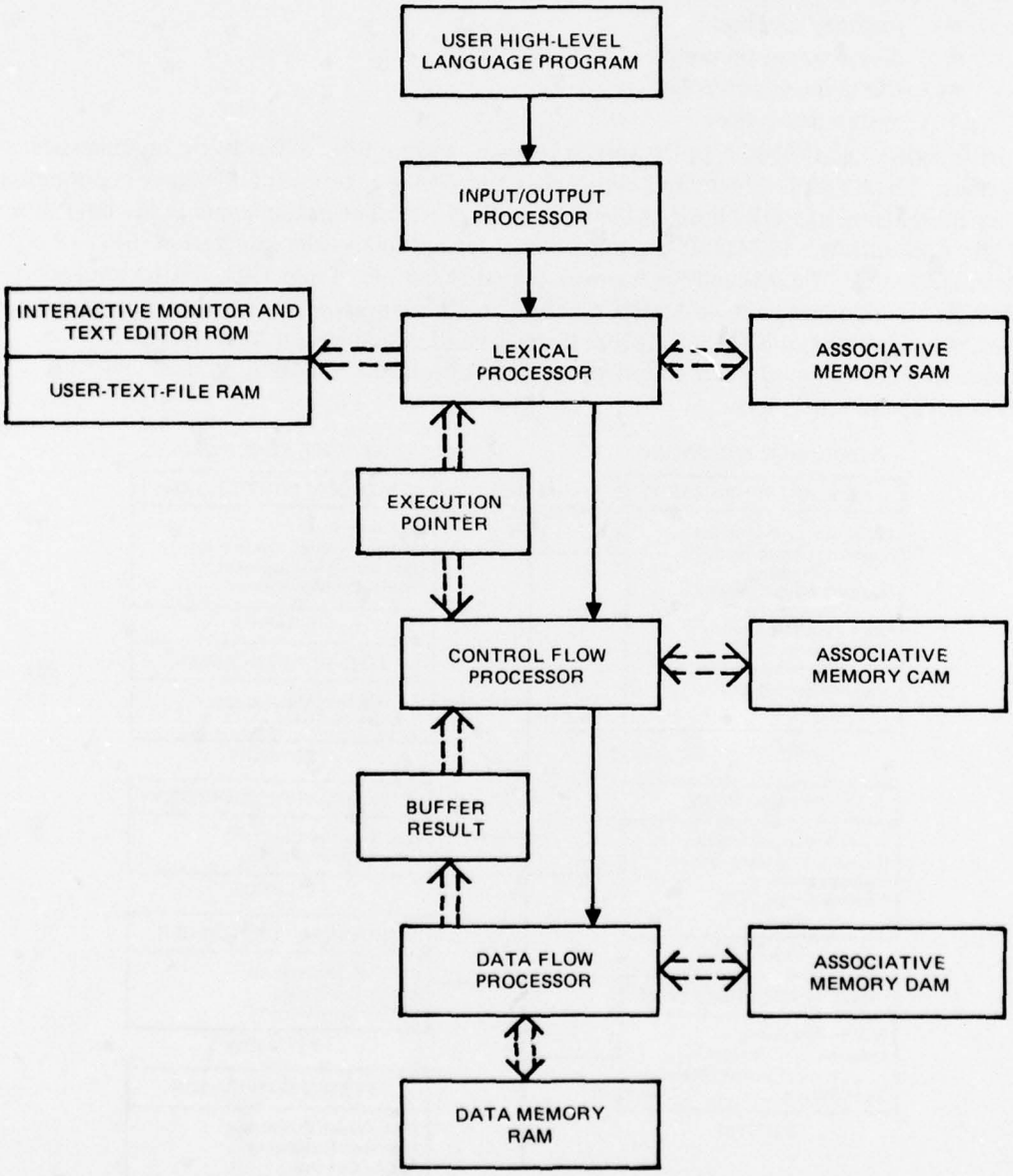



Figure 3. Functional diagram of Chu's ALGOL computer.

Direct execution computer architecture work in the 1960's involved several languages including EULER, FORTRAN, and PL/1. A significant contribution reported in 1971 was the design and construction of the SYMBOL computer system (12) under the direction of R. Rice and W. R. Smith, both employed by Fairchild at the time. The SYMBOL architecture consisted of several processors interconnected with a main bus and incorporated a virtual memory allocated automatically by hardware. The main processing parts included:

- central processor
- translator
- interface processor
- memory controller
- memory relaimer
- disk channel processor
- I/O channel controller
- system supervisor

Figure 4 shows the SYMBOL processors, functions, and number of hardware modules per processor. The SYMBOL high level language is a free format procedural language containing the useful features of FORTRAN, ALGOL, and PL/1. Absent in the language are data type and size declarations. SYMBOL was implemented using small-scale-integration (SSI) 12 X 17 inch single layer PC boards and represents a graphic example of late 1960 SSI technology. SYMBOL was donated to Iowa State University in 1971 and has been used as a research tool. One of the criticisms of SYMBOL is that language and system extensions are difficult to implement since the modules are hardwired. In addition, the SYMBOL system executes only the SYMBOL language.

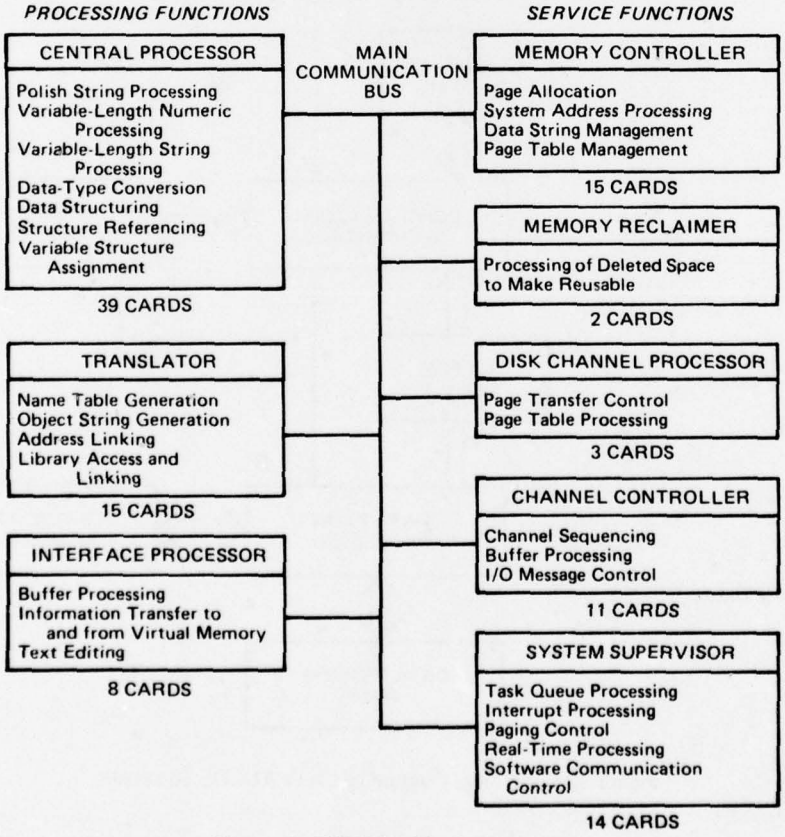



Figure 4. SYMBOL block diagram.

Several APL designs have appeared (13, 14, 15) and IBM currently offers the 5100 desk-top computer that executes APL or BASIC at the flick of a front panel switch. Other direct execution computer architectures which have emerged are SNOBOL 4, HYDRA, PL/1, and IPL processors (9). In 1967 Hawryszkiewycz reported on a microprogrammed system which simulated analog computer problems (16). At least one interactive simulation language based on an interpretive translator has appeared, however, the interpreter is written in mini-computer assembly language and not microprogrammed (17). General Electric has developed a set of hardware microprocessor modules to directly execute high level statements for realtime signal processing (18).

The sampling of direct execution computer architecture designs reviewed here indicates a demonstrated feasibility for directly executing a high level language in hardware and/or firmware. The literature shows that direct execution computer architectures have been applied to both general purpose and specific problem-orientated languages. It should be noted that the more recent designs (post 1970) have used microprogramming as the method of implementing the functional elements of a direct execution language. This approach is an advantage to the designer to produce flexible firmware which can be easily modified during development and extended when future language expansion is required. In addition, a direct execution system and language eliminates the need for multiple system programs, such as compilers, assemblers, and linking loaders. As a result, the user interfaces only with the high level language and its execution and does not need to know the details of such system programs. The designs reviewed here were not economical to build in the technology of the 1960's and early 1970's. However, current large-scale-integration, microprocessor, and memory technology make implementation feasible and cost effective. These advantages and others continue to emerge as research in direct execution architectures and directly executable high level languages progresses.

## DIRECT EXECUTION COMPUTER ARCHITECTURES FOR SIMULATION

The relationship between direct execution computer architecture concepts and continuous system simulation should be examined closely. In other words, why apply direct execution to simulation and at what level should it be applied? It is a generally accepted fact that direct execution systems improve the interface between user, programming language, and application. Interactive model development and problem solution are particularly important in continuous system simulation since these factors eventually influence designer efficiency and costs. A direct execution system allows a simulation language user to interactively develop models since there is no need to wait for the output of chained system programs. This environment means less software layers between the user and equipment. Large-scale-integration implementation of a direct execution system provides a low-cost high-performance alternative to existing digital and analog-based simulation systems. A microprogrammed direct execution architecture using bipolar technology parts can be optimized to perform specific functions required in continuous system simulation problems. Such an optimized combination enables basic operation times as low as 200 nanoseconds to be achieved and provides fast execution speeds for realtime simulation and control. The low cost characteristics of large-scale-integration parts translates to low system costs and makes possible the use of multiple direct execution systems in distributed simulation system architectures. This would include the partitioning of large problems into sections executed by a direct execution system and the application of parallel processing integration algorithms (19, 20). Given that these potential advantages exist, the levels at which direct execution concepts can be applied to continuous system simulation must be explored.

First, let us examine the different phases of interactive continuous system simulation. Figure 5 shows the processes involved in simulating dynamic systems in an interactive environment. User interaction with the simulation system occurs at two distinct levels: (1) problem preparation and documentation, and (2) model description and simulation. Each of these levels includes the several related functions listed here:

1. problem preparation and documentation
   a. text editing modification and storage
   b. report preparation
2. model description and simulation
   a. model dynamics
   b. initial conditions
   c. procedural and solution control sections
   d. functions and subroutines
   e. run-time execution
   f. run-time I/O and display
   g. parameter modification
   h. post-run display, plotting, and processing

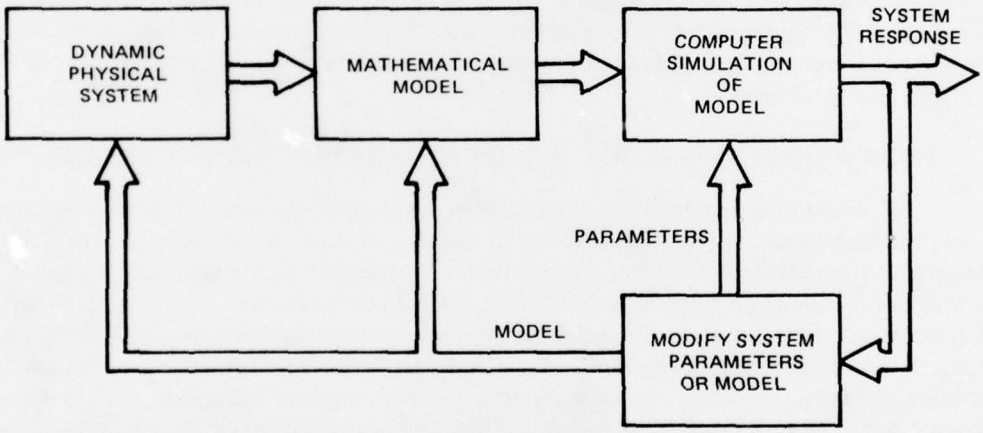



Figure 5. Continuous system simulation process.

A third simulation level includes a realtime environment where the simulation system is dedicated on-line with other computer equipment or actual system hardware. Although user interaction is minimal, the direct execution architecture must account for this mode of operation.

Figure 6 shows the various phases listed above partitioned into a direct execution architecture where each section performs a specific function. The text editing, modification, and storage operations are easily incorporated by a direct execution processor. Since user response is slow during these phases, the processor can perform a lexical scan and syntax

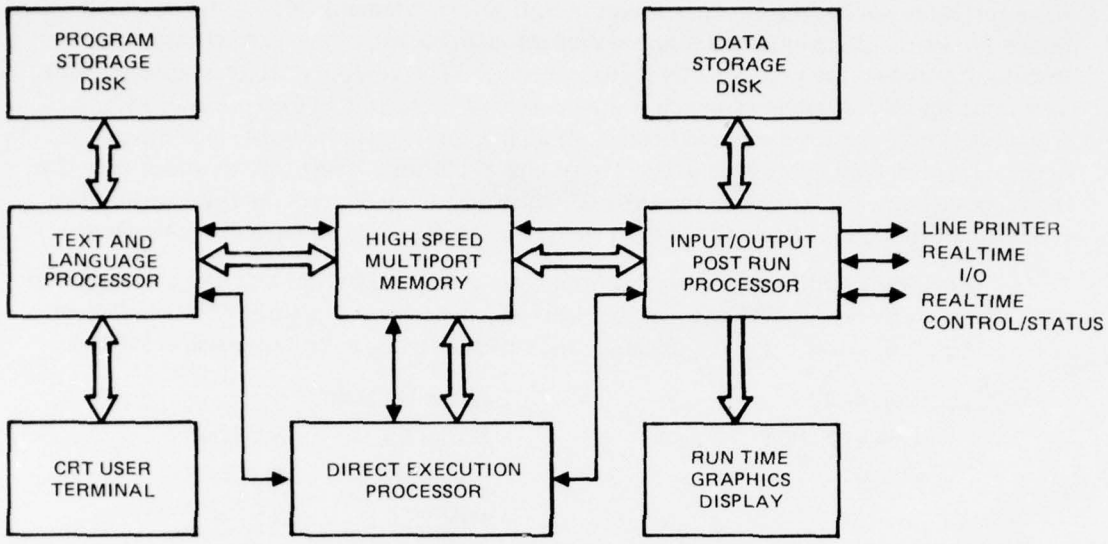



Figure 6. Direct execution computer architecture for interactive continuous system simulation.

check on each line as entered by the user. The text and language processor can perform the storage and retrieval of prepared text on a disk system. As the user enters statements for model dynamics, initial conditions, procedural sections, functions, and subroutines, the same processor sets up data structures, stacks, and control for the problem. (This translation process is further discussed below.) These items are stored in multiport memories accessible by other functional processors.

Run-time execution involves the solution of the dynamic system equations using standard numerical integration techniques for differential equations. The problem state variables are updated using the state derivatives and supporting equations as defined by the user. This set of equations is translated into an executable form in the multiport memory. A direct execution processor can perform the run-time solutions, retrieving and storing data from the multiport memory. A separate I/O and post-run processor can receive interrupts from the direct execution processor at communication interval times (usually a subset of the solution-point times) so that data can be displayed or stored during run-time. Realtime data and control can be handled by the I/O and post-run processor. Post-run display and processing are also performed by this processor by retrieving the run-time stored data from a high speed disk. The user-defined operations to be performed during this phase are also stored in the multiport memory.

The translation process performs two main functions. First, an interpreter processes all language statements for lexical and syntax correctness. The interpreter directly sets up

initial conditions by assigning memory addresses and initializing these locations with data. The interpreter can also process parameter modification statements after a run. Second, model dynamics, procedural and solution control sections, functions, subroutines, and post-run statements are processed by a mini-compiler. The compiler places an intermediate code in the multiport memory which is then executed by the direct execution processor. A compilation process is used here (instead of an interpretive translation) since the entire model dynamics section must be present to obtain a solution. Using this approach only the initial condition section must be retranslated when making parameter modifications When a model modification is made the entire problem must be reinterpreted and recompiled.

The initial conditions are most conveniently expressed by the user using an equation-type format. The model dynamics can be expressed in terms of equation or block-diagram oriented language constructs. A typical second-order system can be expressed as follows:

1) Equation

$X' = -A*X - B*Y$

$Y' = X$

2) Block-Diagram

| | |
|---|---|
| MULTIPLY | $S = X*A$ |
| MULTIPLY | $Q = Y*B$ |
| INVERT | $Z = -S$ |
| SUBTRACT | $P = Z-Q$ |
| INTEGRATE | $X = \int P*G1$ |
| INTEGRATE | $Y = \int X*G2$ |

The model dynamics format described here determines the type of processing that is used by the compiler. For example, both formats can be easily translated into a reverse Polish notation. Another approach is to use an intermediate threaded-code that links the current operation to the next and passes parameter addresses and data through stacks and lists. In each case the intermediate code is executed by the direct execution processor.

Figure 6 presents one partitioning scheme for a direct execution architecture for simulation. The various phases of interactive simulation presented in the paragraphs above indicate that direct execution concepts can be applied to the problem preparation and documentation sections. The model description and translation phases can be performed by a similar processor using interpretation and compilation techniques. Run-time execution and I/O operations also can be performed by a direct execution processor. The following sections describe the sample high level simulation language used in this study and a large-scale-integration implementation of a direct execution system for simulation.

## DEMONSTRATION VEHICLE: MICRODARE LANGUAGE

A high level simulation language, MICRODARE, developed by Korn (21) and Conley (22) at the University of Arizona was chosen as the vehicle for demonstrating a direct execution simulation system in this study. MICRODARE is the newest of a line of DARE simulation languages developed under Korn (2). Appendix A summarizes the DARE languages which include both fixed-point (block-diagram) and floating-point (equation) systems that run on a variety of large mainframe computers and minicomputers. Most of the DARE systems are interactive and use translators, assemblers, compilers, and linking loaders to construct a run-time object program.

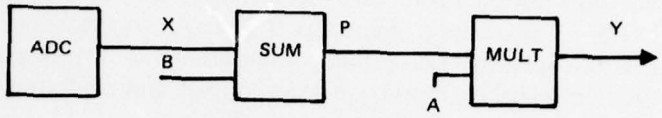
MICRODARE uses system software derived from BASIC and does not require an assembler, compiler, or loader. The language provides interactive program entry, editing, file manipulation, and multi-run control. The two main sections of the MICRODARE language include: (1) problem initialization and run-time control and (2) problem specification via mathematical and realtime operators. Operators are implemented in efficient assembly language reentrant threaded-code subprograms built into the MICRODARE system software or in LSI-11 microcode. Appendix B shows the operators currently available. The user specifies the operators (in correct computational order) for problems in dynamic system simulation, signal processing, process control, and laboratory instrumentation.

MICRODARE uses three formats of data words: (1) 48-bit floating-point, 32-bit mantissa; (2) 16-bit fixed-point fractions, and (3) 16-bit 2's complement integers. The floating-point data is a special format used in the BASIC-dialect portion of MICRODARE. Run-time variables are converted to fractional fixed-point before a run and reconverted to floating-point after the run. MICRODARE also provides for run-time display of problem variables in time or phase-plane formats. Additional information on MICRODARE is available from the references (2, 21, 22).

MICRODARE was chosen as a demonstration vehicle for direct execution for simulation for several reasons. First, the system software was available and executing on commercial PDP-11 and LSI-11 hardware and no system software development effort was required. Second, MICRODARE executes on LSI-11 microcomputer systems and provides a low-cost system (under $12K) with graphics and a capability for special purpose hardware interfaces. Also, the LSI-11 bus is well documented and can be easily interfaced to custom devices. Third, the MICRODARE language constructs are simple and implemented in modular system software that can be easily modified. Fourth, MICRODARE language constructs allow operations in a realtime signal processing or simulation environment. In summary, the MICRODARE language provided an excellent opportunity to investigate and demonstrate the application of direct execution computer architectures without having to invest a large amount of time and effort in starting from scratch to develop new language specifications.

The MICRODARE system software used in this project contains a modified BASIC interpreter and mini-compiler that generate a threaded-code list of block operators and argument addresses. This list is operated on the LSI-11 which executes preprogrammed PDP-11 machine language instructions instructions for each operator. Figure 7 shows a simple data acquisition and processing example and the threaded-code address list generated by the MICRODARE mini-compiler. An index register pointer is used as a "virtual program counter" and incremented after each memory reference. Each canned operator subprogram contains a preindexed, indirect, and increment jump instruction which provides the linkage mechanism to the next operator. In this way, fast executing subprograms are efficiently and simply linked together.

```
ADC      X          : *READ A-D CONVERTER
SUM      P = X + B  : *ADD B TO X
MULT     Y = P * A  : *PERFORM MULTIPLICATION
```




THREADED-CODE ADDRESS LIST

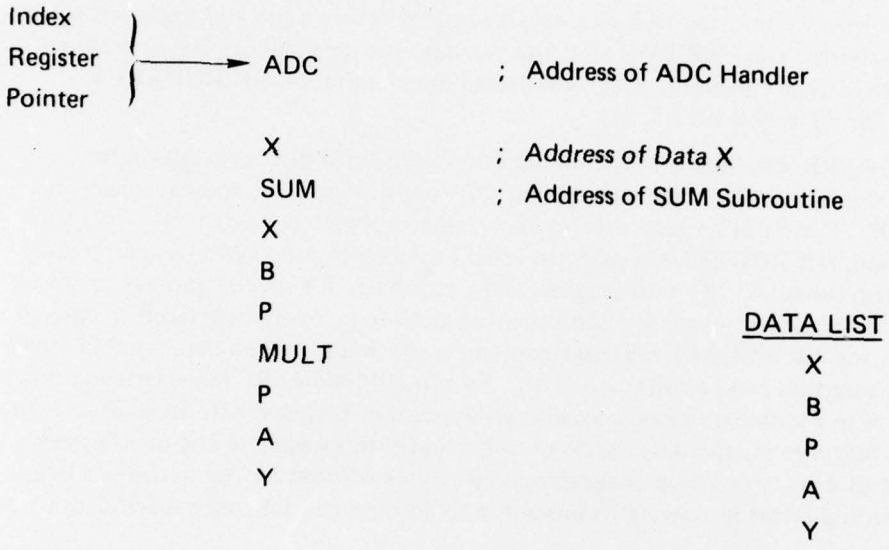



Figure 7. MICRODARE data acquisition example.

Figure 8 shows an interactive problem simulating a servo mechanism and a mean-square-error computation. Appendix C contains additional and more complex MICRODARE sample problems including a two-dimensional torpedo dynamics simulation.

$$E = U - X \qquad \text{(servo error)}$$

$$\left.\begin{aligned} \frac{dX}{dT} &= G1\% * XD \\ \frac{dXD}{dT} &= G2\% \,(H*E - R*XD) \end{aligned}\right\} \qquad \text{(servo state equations)}$$

$$P = G3\% \int_0^T E^2 \, dT \qquad \text{(Mean square error)}$$

```
10  ***** ********************  A SIMPLE SERVO SIMULATION
15  CLEAR STACK
20  DT=0.005  :  TM=0.9999 :  ***** SIMULATION PARAMETERS
30  G1%=20  :  G2%=40  :  G3%=10  : *****INTEGRATOR GAINS
40  X=0  :  XD=0  :  P=0  : ***** INITIAL VALUES OF STATE VARIABLES
45  H=0.375  : ***** A CONSTANT PARAMETER
50  U=0.8  : ***** INPUT STEP VALUE
55  PRINT "ENTER DAMPING PARAMETER R"
60  INPUT R  : ***** INPUT INTERACTIVELY REQUESTED FROM USER
65  T=0
70  DRUN : ***** DIFFERENTIAL-EQUATION-SOLVING RUN
80  END
90  ***** ********************  BLOCK-DIAGRAM PROGRAM FOLLOWS
20100  DIF E=U-X
20200  MULT RX=R*XD
20300  MULT HE=H*E
20400  DIF Y=HE-RX
20500  MULT ES=E*E
20550  ***** ************ RUN-TIME DISPLAY
20560  DISPT X
20570  DISPT E
20580  ***** ************ INTEGRATORS ARE LAST!
20600  INTG XD(Y*G2%
20700  INTG X(XD*G1%
20800  INTG P(ES*G3%  : ***** MEAN-SQUARE ERROR
```

Figure 8. MICRODARE servo mechanism example.

## LSI IMPLEMENTATION OF DIRECT EXECUTION MICRODARE

A functional partitioning scheme of an interactive simulation system was briefly depicted in Figure 6. Depending on the function of each block, one can assign various large-scale-integration devices and subsystem hardware, firmware, and software to perform the specific tasks. Figure 9 shows the partitioning scheme as applied to the MICRODARE simulation language. A key feature of the scheme used here was the separation of the problem preparation and translation functions from the run-time execution of the problem. The former requires moderate execution speed while the latter requires high speed.

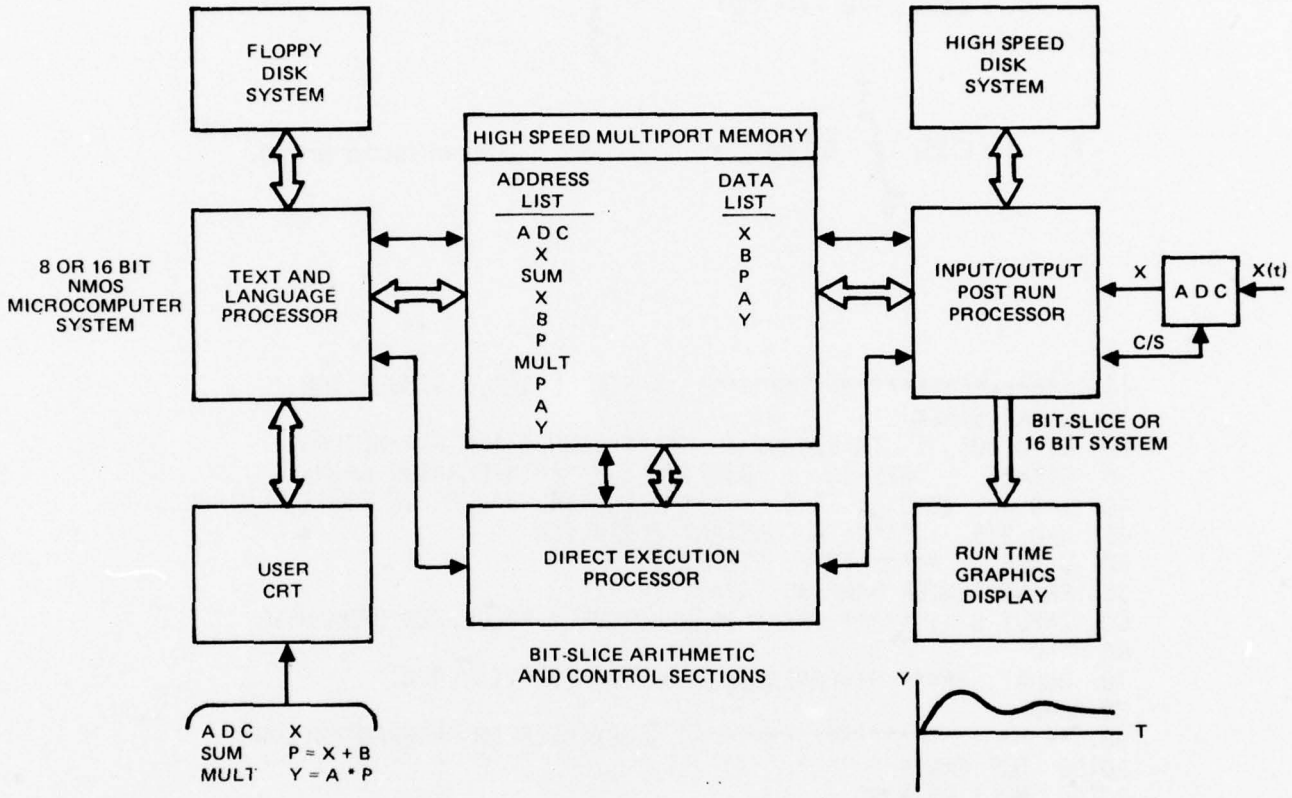



Figure 9. Direct execution architecture for MICRODARE.

Low cost and commercially available large-scale-integration devices and subsystems were considered and included three component types: (1) 8 and 16-bit microprocessors; (2) microprogrammed bit-slice devices; and (3) high speed memory. The blocks in Figure 9 are repeated here with the type of large-scale-integration devices applicable to each subsystem.

### 1. Text and Language Processor

Eight or 16-bit n-channel metal-oxide-semiconductor microprocessor with local program read-only-memory and data random-access memory. Interfaces to user terminal, program storage floppy disk subsystem, and multiport memory. Moderate speed microprocessor (200–300 kilo-operations-per-second) is required since user interface is typically at human response times. This processor also performs the interpreter/compiler functions.

**2. Multi-Port Memory**

High speed (50–200 nanosecond access time) memory configured with ports to the text and language processor, direct execution processor, and input/output and post-run processor. The direct execution processor should have priority over the other two since it will execute much faster. However, in a realtime environment, the input/output processor might require a high priority for rapid data transfers. Memory size should be approximately 16–32K of 16–32 bit words.

**3. Direct Execution Processor**

Bipolar bit-slice components including register, arithmetic and logical units, microprogram memory, microprogram sequencer unit, and miscellaneous circuitry. This processor must have the capability to perform microprogrammed operations such as multiplication, summation, and integration. Special functions components such as a multiplier unit and sine/cosine read-only-memories could be used here.

**4. Input/Output and Post-Run Processor**

High speed 16-bit microprocessor or bipolar bit-slice components with architecture featuring multiple input/output ports to memories, realtime devices, and high speed storage and display peripherals. Post-run processor can be programmed via the multiport memory with instructions supplied by user.

To fully implement the four major sections described above using a variety of large-scale-integration devices represented a monumental task beyond the time and finances available for this project. Therefore, a simplified architecture scheme was used that still demonstrated direct execution concepts for simulation. The simplified architecture consists of a dual processor system interconnected via a dual port memory as shown in Figure 10. The functions of the text and language processor and the input/output processor were combined and implemented on a LSI-11 microcomputer system. A dual double-density floppy disk subsystem provides program and run-time storage. A Tektronix 4006 graphics terminal provides program entry and run-time displays. The dual port memory (16K X 16) is part of the total (28K X 16) LSI-11 memory. The MICRODARE system software is stored on the floppy disk system and executes in the LSI-11 and dual port memory. In addition, a bit-slice microassembler is run on the LSI-11 to develop microcode software for the direct execution processor (23).

In the dual processor architecture depicted in Figure 10, the user inputs the MICRODARE problem via the Tektronix 4006 terminal. The MICRODARE system software translates the problem into an intermediate threaded-code that consists of an operation and data address list for the dynamics state variable computations and numerical integration. The address list and corresponding data locations are stored in the dual port memory. Once the dual port memory contains this threaded-code, the LSI-11 begins the simulation run. The LSI-11 signals the direct execution processor to start its processing. The direct execution processor fetches the operation opcodes and data operands, performs the operation, stores the result, and links to the next operation. After the last operation is performed, the direct execution processor returns control back to the LSI-11 which performs numerical integration of the stated variables and displays the selected variables on the graphics terminal. The LSI-11 also performs simulation-study control and pre-run and post-run computations. Eventually, the direct execution processor will perform all of the numerical integration and realtime input/output. The following sections further describe the components and operation of the dual port memory, direct execution processor, and writeable control store and their interaction with the LSI-11 as shown in Figure 11.

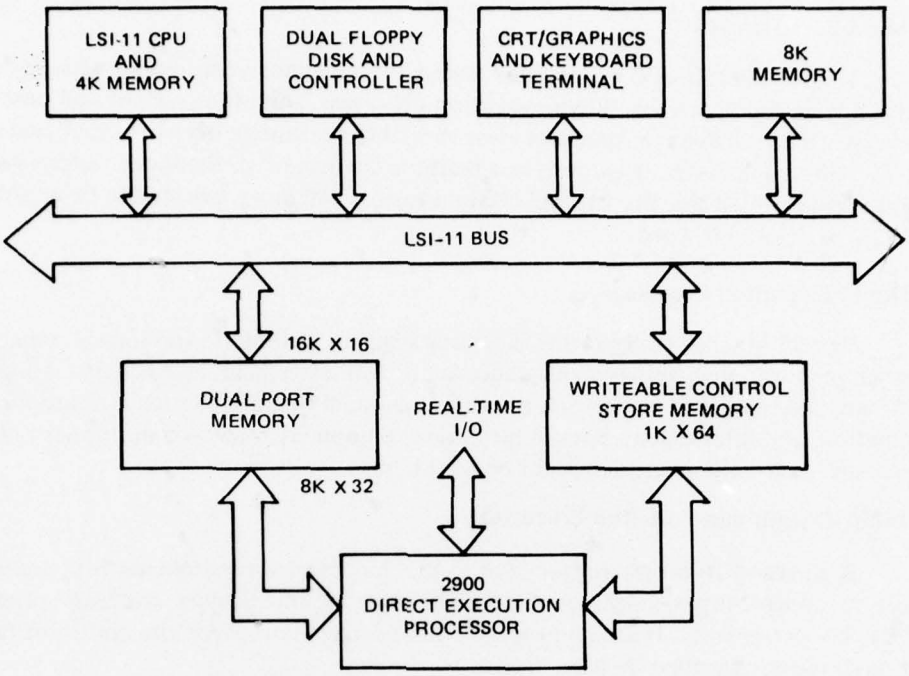



Figure 10. MICRODARE dual processor system.

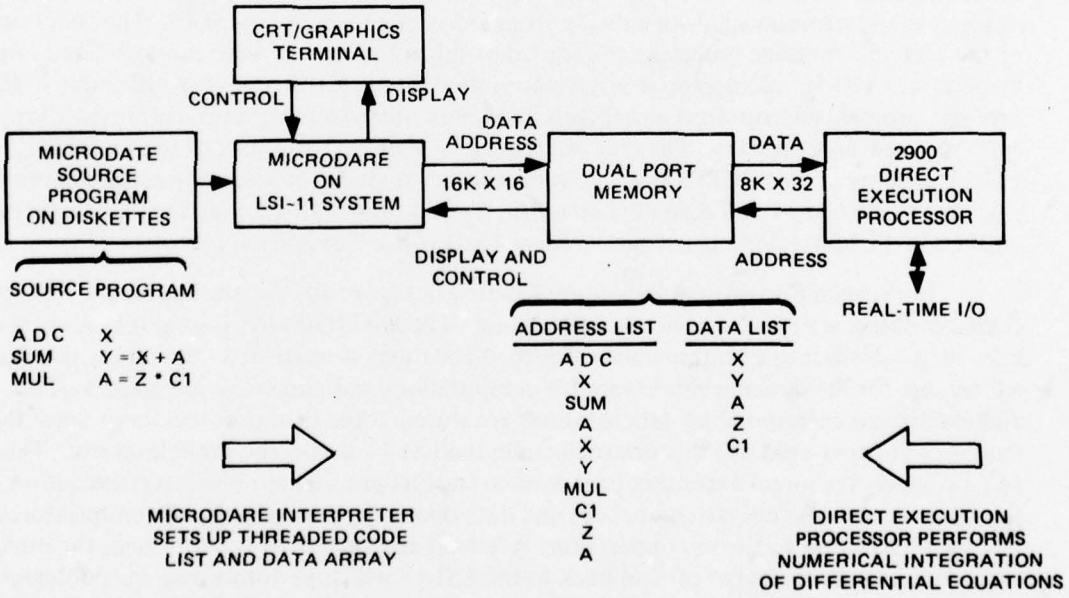



Figure 11. Dual processor operation for MICRODARE.

The dual port memory provides a path of communication between the LSI-11 and the direct execution processor (DEP). As seen by the LSI-11, the dual port memory looks like an ordinary LSI-11 16K memory circuit board. As seen by the DEP, the dual port memory looks like an 8K word by 32-bit memory. The memory is electrically configured as an array of 8K X 32-bit words with multiplexers to enable reading and writing 8-, 16-, or 32-bit words. This feature is required because the LSI-11 must have 8- or 16-bit read/write capability while the DEP requires 16- or 32-bit read/write capability. The electrical specifications governing the use of the LSI-11 port are identical to the normal "Q-BUS" specifications found in the LSI-11 technical manual (24). The DEP port, however, is not standard and consists of the following:

1. 13 address lines (ADD)
2. 32 data lines (bidirectional)
3. A write (read not) command line (W)
4. A half (whole not) word command line (HW)
5. A memory request line (MR)
6. A memory acknowledge line (from memory) (RA)
7. A read data ready line (DR)

Note: The above lines are low true.

The operation of the DEP port is as follows, and its timing diagram is shown in Figure 12.

1. Read Operation
   - The DEP asserts the address (ADD) and negates the write (W) and half work (HW) signals.
   - Memory request (MR) is then asserted.
   - The memory sends request acknowledge (RA) when the DEP port gains access.
   - After the read access delay the data is placed on the lines.
   - The true to false transition of data ready (DR) should be used to strobe the data.
   - The DEP then negates memory request (MR).
2. Write Operation
   - The DEP asserts the address and write signal (W) and negates the half word (HW) signal.
   - Memory request (MR) is asserted.
   - The memory sends request acknowledge (RA) when port access is granted.
   - The DEP places write data on the data lines.
   - After data has been stable on the lines for 10 nsec the write signal (W) is negated (assertion can happen anytime).
   - Memory request (MR) is then negated.

Assertion of the half word command (HW) causes the above operation to be performed on a half rather than whole word. When half word mode is selected, the least significant address bit indicates which half of the word is to be used. This is also the convention followed in byte mode using the LSI-11 port.

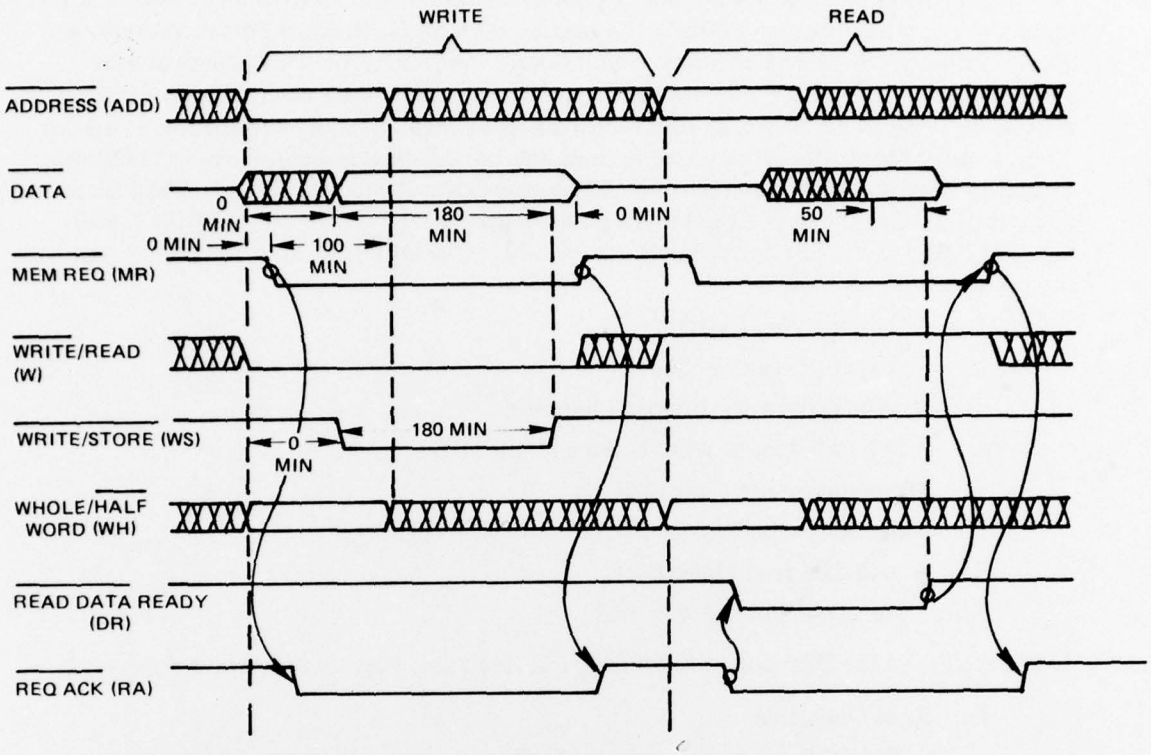



Figure 12. Dual port memory timing diagram (DEP port).

The direct execution processor (DEP) functional block diagram is shown in Figure 13 and is implemented using the Advanced Micro Devices 2900 bit-slice family of parts (24). The DEP consists of a data processing section and a microinstruction sequencer section. These sections include the following functional parts:

1. 32-bit data read/write interface to dual port memory.
2. 32-bit AMD 2903 register, arithmetic, and logical unit (RALU).
3. 16 X 16 TRW multiplier unit.
4. 16-bit AMD 2930 memory address controller.
5. AMD 2910 microsequence controller.
6. 1K X 64-bit writeable control store memory and pipeline register.
7. 10-bit operation code (instruction) register.
8. 32-bit memory data register with selectable 16-bit upper and lower sections.
9. 16-bit memory address register to dual port memory.
10. Status register and condition code multiplexer.
11. Direct data paths from memory data register to memory address register.
12. Data path from pipeline register to 32-bit RALU.
13. System clock circuitry, approximately 5 MHz.

14. Control lines to dual port memory.
15. Microprograms for MICRODARE operations.

These parts are wire-wrapped on two quad-type boards similar to LSI-11 boards.

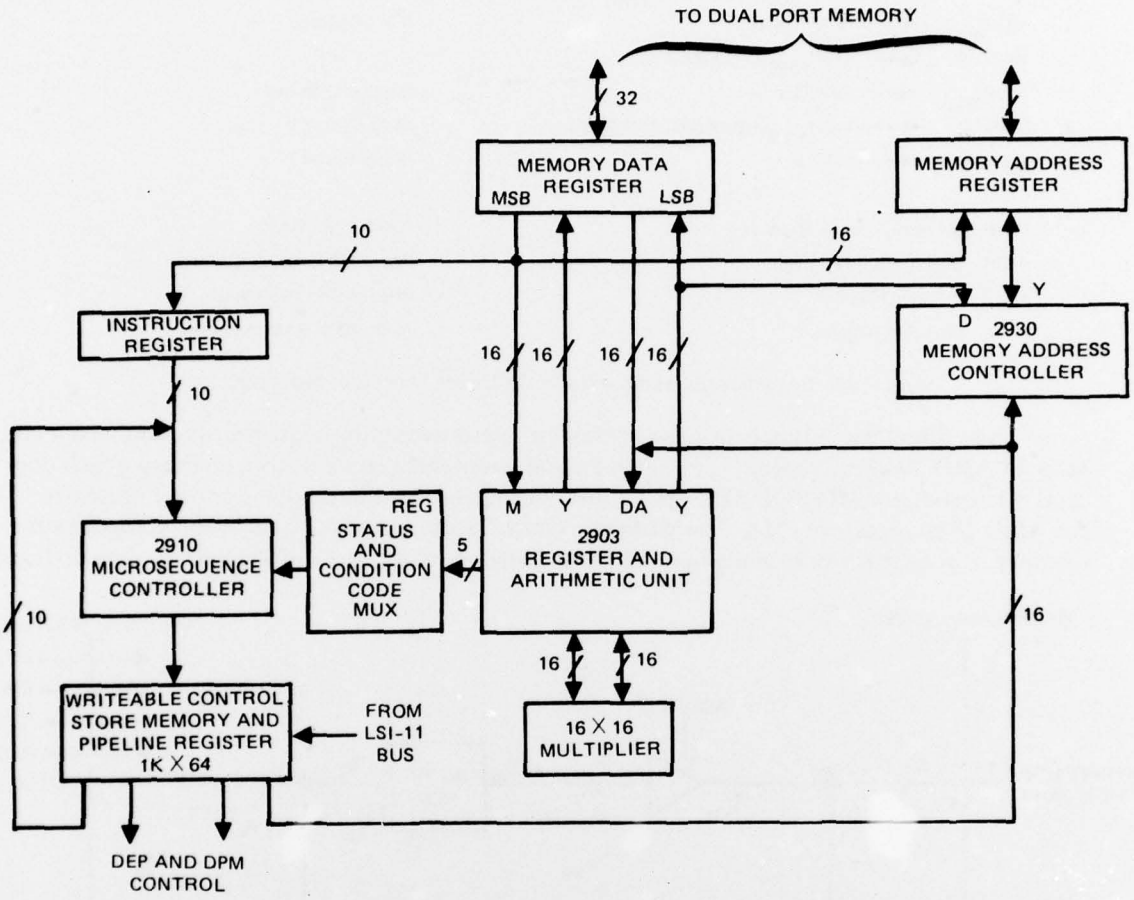



Figure 13. Direct Execution Processor (DEP) functional block diagram.

The LSI-11 and direct execution processor communicate through an absolute "mailbox" address virtual program counter (VPC) in the dual port memory. The direct execution processor periodically examines the VPC location, commences its operation if the LSI-11 program has set the word to a microcode address, and puts the LSI-11 in a wait mode. Figure 14 shows the sequence for an ADD operation by the direct execution processor. For details, the reader is referred to the internal architecture of the AMD 2903 and 2930 parts (25, 26). Notice that the 32-bit interface to the dual port memory allows reading of (1) an operation code and data address or (2) two data addresses from the threaded-code list. Data is read and stored in the memory as 16-bit 2's complement fixed-point fractions. All MICRODARE operations, except MULTIPLY, are performed by the 2903 RALU. When the operation is complete, the direct execution processor stores the update virtual program counter in VPC and enters a wait loop. The LSI-11 reads this address in VPC as a threaded-code address and continues with the display operations. Further details and speed benchmarks will be documented in future reports.

| | | | |
|---|---|---|---|
| 1. | MAR | ← PC | /Address of Operator/Data Address/Pair |
| 2. | MDR | ← DPM(MAR); PC ← PC + 2 | |
| 3. | IR | ← $MDR_{MSB}$; PUSH $MDR_{LSB}$; MAR | ← PC/ |
| 4. | MDR | ← DPM(MAR); PC ← PC + 2 | /Address of Data Address Pair |
| 5. | MAR | ← $MDR_{MSB}$; PUSH $MDR_{LSB}$ | |
| 6. | $MDR_{MSB}$ | ← DPM(MAR) | /First Operand |
| 7. | RAM(B) | ← $MDR_{MSB}$; POP MAR | |
| 8. | $MDR_{MSB}$ | ← DPM(MAR) | /Second Operand |
| 9. | RAM(A) | ← $MDR_{MSB}$ + RAM(B); POP MAR | /Add Operands |
| 10. | DPM(MAR) | ← RAM(A) | /Store Result |

MAR: Memory Address Register
MDR: Memory Data Register
IR: Instruction Register
DPM: Dual Port Memory
RAM: 2903 RAM
PC: 2930 Program Counter
PUSH/POP: 2930 Stack
A/B: 2903 RAM Address

Figure 14. Sequence for ADD operation in Direct Execution Processor.

The direct execution processor contains a microinstruction sequencer section which contains an AMD 2910 microsequence controller and a writeable control store memory which contains microcode for MICRODARE operations. Details of the microsequencer are referred to the AMD 2910 literature (25). The writeable control store board consists of the control store memory, the system clock, and a parallel I/O interface to the LSI-11. Figure 15 is a block diagram

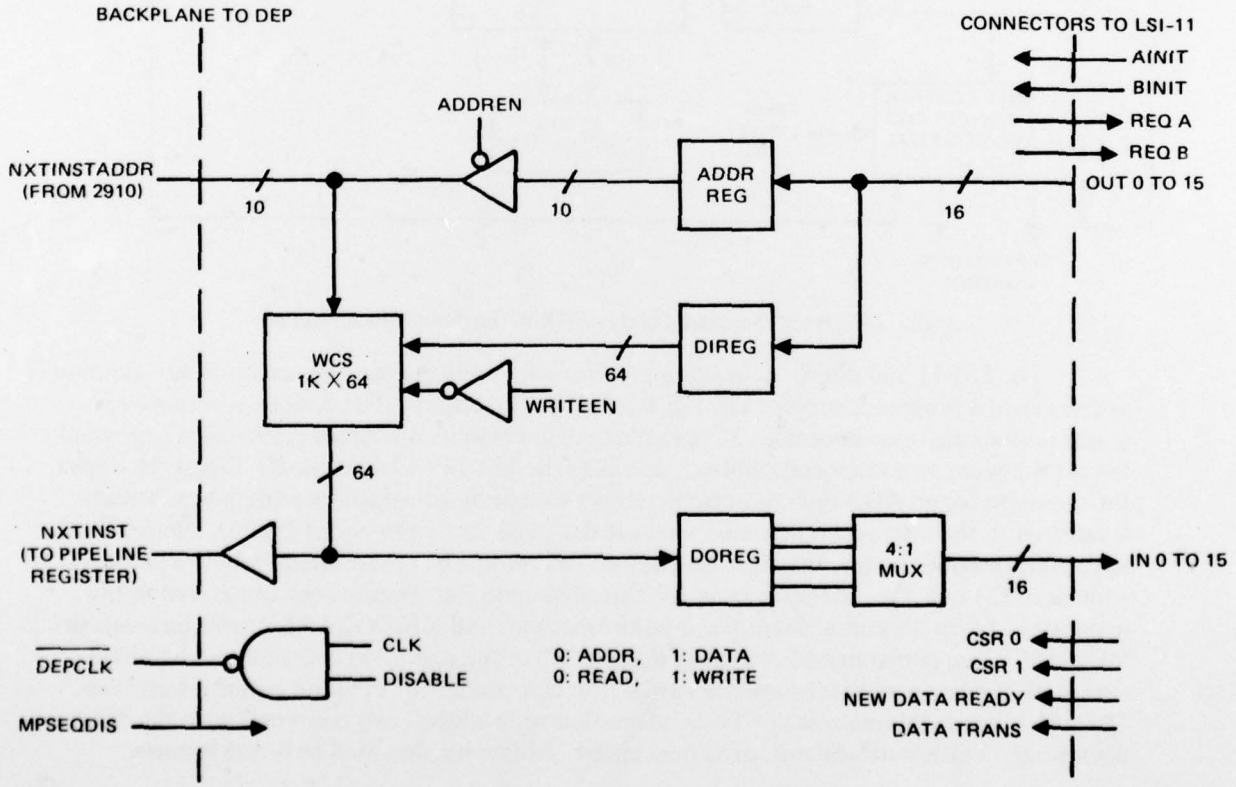



Figure 15. Writeable control store block diagram.

of the board. The control store consists of 1K × 64 bits of high speed RAM. Ten address lines, NXTINSTADDR, from the backplane select the next microinstruction, NXTINST, which is buffered onto the backplane. Access time of the RAM is 45 ns. Also supplied to the backplane is the 5.0 MHz system clock and a disable signal, MPSEQDIS. When MPSEQDIS is high, the writeable control store is being addressed by the LSI-11 and NXTINSTADDR should be disabled (high impedance state).

The LSI-11 can read or write the control store via a DRV-11 parallel interface. Control store addresses and data are transmitted in successive 16-bit words, one word for the address and four words for a complete microinstruction. The two control lines, CSR0 and CSR1, from the DRV-11 are used to direct the I/O. CSR0 describes the contents of the data lines, OUT 0TO15. CSR0 low indicates control store address and initiates the read/write sequence; high indicates control store data. The least significant 16 bits are transmitted first. Low CSR1 at the time the address is loaded (CSR0 low) directs the logic to load the contents of the control store into the output register, DOREG, for subsequent multiplexing onto the IN 0TO15 lines to the DRV-11. A high CSR1 indicates a write sequence which is initiated after the input data register, DIREG, is loaded. REQB is used as an interrupt signal for interrupt driven I/O by LSI-11. An interrupt is generated by the completion of write or read cycle. REQA provides a ready signal for the LSI-11. It is reset during access to the control store. AINIT or BINIT initializes the I/O logic.

The LSI-11 microcomputer system shown in Figure 10 is an off-the-shelf system which executes the MICRODARE system software. The LSI-11 has a 28K 16-bit word memory space, 16K of which is the dual port memory. Other functional elements of the LSI-11 system include:

1. LSI-11 microcomputer module with extended instruction set and floating point chips.
2. 28K (16-bit) semiconductor memory, 16K dual port to DEP.
3. Dual double density floppy disk drive/controller with UNIBUS/LSI-11 converter interface.
4. Multi-slot backplane and power supply.
5. RT-11 floppy disk operating system and MICRODARE system software.
6. EIA RS-232 serial interface module to terminal.
7. Panel, clock, and terminator module.
8. Graphics terminal with keyboard entry.
9. Decwriter III line printer/terminal.
10. Parallel interface module to writeable control store memory.

The memory layout for the dual processor system is shown in Figure 16. Additional details on the MICRODARE systems software can be found in the references (21, 22).

The RT-11 operating system resides on diskettes for a double density dual floppy disk system. RT-11 includes a monitor, PIP utilities, text editor, microassembler, BASIC, FORTRAN compiler, linking loader, and device handlers. The application programs used here were MICRODARE simulation language and AMDASM microassembler. The MICRODARE system is a stand-alone program that can be run by typing R MDTEK on the control

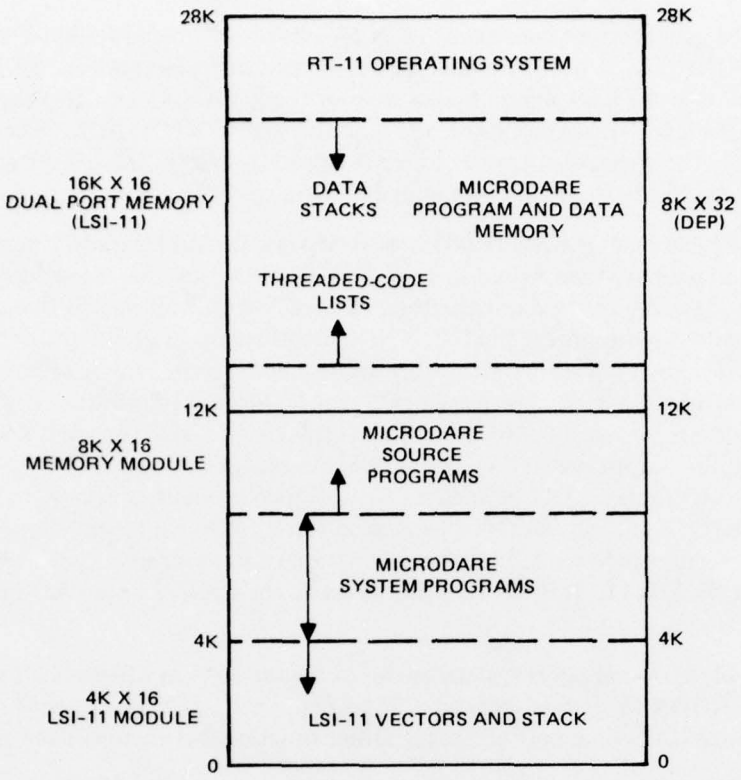



Figure 16. Memory layout for MICRODARE dual processor system.

terminal. MICRODARE is a modified BASIC interpreter and has both file manipulation and text editing capabilities. Some minor modifications to this software have been made (22). The MicroTec AMDASM microassembler program is used to develop microcode for LSI bit-slice devices, such as the 2900 family (23). The program is written in FORTRAN and requires a FORTRAN compiler in order to run on the LSI-11 microcomputer system. The microassembler was installed as three stand-alone programs under RT-11 and can be invoked by typing R AMDASM. AMDASM is used to develop microcode for the direct execution processor. The input to AMDASM is a source file containing microprogram field declarations, variables, and mnemonics. The AMDASM object output is stored on a disk file and later transferred to the writeable control store memory.

## APPLICATION AREAS FOR DIRECT EXECUTION ARCHITECTURES

Simulation of continuous systems is a prime example of an application area for direct execution computer architecture. In an interactive simulation environment, the user must make model and parameter changes to the problem and quickly evaluate the results of the new simulation. He cannot afford (nor his employer) to wait for lengthy compilations and library linkages to produce an executable simulation run. Direct execution computer architectures provide the close relationship between the user and the simulation equipment required in this type of application problem. An additional advantage is increased computation speed since the architecture is optimized to numerically solve systems of differential equations. This advantage becomes important in a realtime data-acquisition or simulation problem where an analog computer is to be replaced. Once the model has been developed

the problem state variables are computed via microprogrammed operations stored in read-only-memory. Clearly such a simulation system has a limited bandwidth depending on the problem time constants and complexity. However, techniques exist to partition systems of differential equations into "slow" and "fast" portions by separation of state derivative equations. In addition, algorithms for parallel solution of differential equations have been investigated (20, 27) and can be adapted into direct execution architectures. In these types of problems, the direct execution architecture described in this paper has extensions to multiprocessor simulation systems.

Direct execution architectures can be applied to problems that require a close user/equipment interface. Laboratory process and instrumentation control is an example where data acquisition, distribution, and signal processing can be performed with a direct execution system. Interactive languages for graphics display systems which require a small amount of data formatting and processing prior to displaying can utilize direct execution concepts. Data base management systems that require information storage and retrieval functions are applications areas of direct execution concepts (28). All of these application areas require subsystem control and data inquiries which can be performed quickly and without recompiling when a model change is made.

The traditional application area for direct execution architectures has been general purpose high level programming languages. Direct execution architectures are continuing to emerge for languages such as BASIC and APL. A stack-oriented 16-bit microcomputer chip set which interprets PASCAL intermediate P-code in hardware has been announced by Western Digital (29). Other semiconductor vendors are reportedly working on similar microcomputer architectures.

## SUMMARY AND CONCLUSIONS

The concepts of direct execution computer architecture for continuous system simulation investigated under this project fall into two categories: (1) interactive development mode and (2) realtime computation mode. The first category involves the high level language issues and the interface to the direct execution hardware. The second category relates to the direct execution architecture and its capability to perform high speed computations.

User requirements in the interactive development mode clearly indicate the advantages of an interpreter-based language over a compiler-based language. The interpreter instantaneously reports back errors to the user as he enters language statements. In the case of MICRODARE, a combination of interpreter and compiler is suitable since the numerical solution of differential equations involves repeated computation of the problem state derivatives. The MICRODARE mini-compiler sets up a threaded-code list for the simulation run which is executed by the microprogrammed direct execution processor. Language extensions to MICRODARE, such as the addition of special purpose mathematical functions, can be made by adding the appropriate microcode or hardware units. Although the MICRODARE interpreter is derived from BASIC, it is coded in PDP-11 assembly language and executes fairly rapidly on a minicomputer (less than 2 seconds for most problems). The MICRODARE interpreter/compiler would execute still faster if some of the assembly language subroutines and functions could be microcoded. The new LSI-11 module with writeable control store memory can provide this feature. Assembly language modifications were made to the interpreter software in this project to accommodate transferring parameters through the dual port memory to the direct execution processor.

The realtime computation mode is necessary if the direct execution system is to be used to solve systems of differential equations on-line. Replacement of analog computer functions in a hybrid simulation system is such an example. In this project, high speed bit-slice devices provided the means of demonstrating the direct execution architecture. The flexibility of these parts enable an optimized design to retrieve, compute, and store simulation variables in the dual port memory. The partitioning scheme used enforced the need for special purpose LSI units, such as multiport memories, multipliers, sine/cosine ROM, memory address controllers, and floating point processors. The units provide high speed processing which would normally be performed by the LSI-11 or in direct execution processor microcode. Additional units to speed up the language interpreter sections could be used if available. The partitioning scheme also pointed out the advantages of independent parallel data and address processing sections in the design. The data processing section in this project used 16-bit fixed point fractional arithmetic. This approach requires the user to scale the simulation problems as in the analog computer style. A floating point architecture would negate this requirement.

Investigations in this project indicated potential future work areas in direct execution architectures for simulation. These areas are summarized here:

1. Multiport Memory LSI Chips – high speed devices to develop low cost memories for interprocessor communication.

2. Floating Point Direct Execution System – enables scale-free simulation problems; separate data and program memories.

3. High Speed LSI Floating Point Processor – multi-chip bipolar or SOS processor to complement floating point system; existing NMOS processor is too slow.

4. Optimized Interpreter for Simulation – translation phase of simulation described earlier in report and is optimized to process simulation statement constructs.

5. Input/Output Interfaces for Direct Execution Process – design with capability to interface to other direct execution processors to handle large simulations.

6. Direct Execution Multiprocessor System Software – system software to handle multiprocessor simulation possibly based on concurrent PASCAL concepts.

7. Special Function LSI Units – high speed function table look-up units, sine/cosine ROMs, exponential and transcendental functions, string processing operators, and memory stacks are examples of these.

## REFERENCES


1. M. L. Mitchell, "PHYSBE in the Raytheon Scientific Simulation Language (RSSL)," SIMULATION, March 1974, pp. 81–86.
2. G. A. Korn and J. V. Wait, *Digital Continuous-System Simulation*, Prentice-Hall, Englewood Cliffs, N.J., 1977.
3. Mitchell and Gauthier Associates, Inc., *ACSL, Advanced Continuous Simulation Language, User Guide/Reference Manual*, Concord, MA, 1975.
4. W. Koral and L. Schirm, "LSI Circuits for Digital Simulation," Proceedings of the 1978 Summer Computer Simulation Conference, Newport Beach, CA, pp. 85–87.
5. Y. Chu, "Direct-Execution Computer Architecture," 1977 IFIP Congress Proceedings, pp. 7–13.
6. Y. Chu, "An LSI Modular Direct-Execution Computer Organization," COMPUTER, July 1978, pp. 69–76.
7. E. Block and D. Galage, "Component Progress: Its Effect on High-Speed Computer Architecture and Machine Organization," COMPUTER, April 1978, pp. 64–76.
8. Y. Chu, Editor, *High-Level Language Computer Architecture*, Academic Press, New York, 1975.
9. C. R. Carlson, "A Survey of High-Level Language Computer Architecture," *High-Level Language Computer Architecture*, Academic Press, New York, 1975, pp. 31–62.
10. J. P. Anderson, "A Computer for Direct Execution of Algorithmic Languages," Proceedings EJCC, 1961, pp. 184–193.
11. Y. Chu, "Introducing the High-Level Language Computer Architecture," Technical Report TR-227, University of Maryland, Computer Science Center, 1973.
12. R. Rice and W. R. Smith, "SYMBOL – A Major Departure From Classic Software Dominated von Neumann Computing Systems," Proceedings SJCC, 1971, pp. 575–587.
13. R. Zaks, "Microprogrammed APL," Proceedings IEEE International Computing Society Conference, 1971, pp. 193–194.
14. S. M. Nissen and S. J. Wallach, "An APL Microprogramming Structure," Proceedings Symposium on High-Level-Language Computer Architecture, University of Maryland, 1973, pp. 43–51.
15. Micro Computer Machines, Inc., *MCM–70 Brochure*, Willowale, Ontario. 1974.
16. I. T. Hawryszkiewycz, "Microprogrammed Control in Problem-Oriented Languages," IEEE Transactions on Electronic Computers, Vol. EC-16, No. 5, October 1967, pp. 652–658.
17. R. D. Benham, "Interactive Simulation Language – 8," SIMULATION, Vol. 16, No. 3, March 1971, p. 116.
18. General Electric/Ordnance Systems, *Hardware Micro-Processor System Guidelines*, Pittsfield, MA, 1975.
19. G. A. Korn, "Back to Parallel Computation: Proposal For a Completely New On-Line Simulation Using Standard Minicomputers For Low-Cost Multiprocessing," SIMULATION, August 1972, pp. 37–45.

20. M. A. Franklin, "Parallel Solutions of Ordinary Differential Equations," IEEE Transactions on Computers, Vol. C-27, No. 5, May 1978, pp. 413–420.

21. G. A. Korn, "MICRODARE: A Fast, Direct-Executing High-Level-Language System for Small Computers," Technical Report, University of Arizona, Electrical Engineering Department, June 1978.

22. S. W. Conley, "Software Design for Simulation and Instrumentation," Ph. D Thesis, University of Arizona, Electrical Engineering Department, 1979.

23. Microtec, *Meta Assembler Manual for AMD 2900 Microprocessor,* Sunnyvale, CA, 1978.

24. DIGITAL Equipment Corporation, *Microcomputer Handbook,* Maynard, MA. 1977.

25. Advanced Micro Devices, *Am 2903 Four-Bit Slice and Am 2910 Controller,* Sunnyvale, CA. 1978.

26. Advanced Micro Devices, *Am 2930 Memory Address Controller,* Sunnyvale, CA, 1977.

27. W. L. Miranker and W. Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equations," Mathematical Computing, Vol. 21, 1967, pp. 303–320.

28. D. L. Small and D. O. Christy, "Command Center Information System," NELC TD 498, San Diego, CA, November 1976.

29. J. G. Posa, "Microcomputer Made for PASCAL," Electronics, 12 October 1978, p. 155.

## APPENDIX A

Table A1. DARE languages summary.

| Feature \ DARE System | DARE-I 1969 | DARE-II 1970 | DARE-IIIB 1971 | DARE P 1973 | DARE/ELEVEN 1974 | MICRODARE 1977 |
|---|---|---|---|---|---|---|
| Computer System | DEC PDP-9 | DEC PDP-9 | CDC 6400/ DEC PDP-9 | CDC 6400, IBM 360, UNIVAC 1110, etc. | PDP-11 Family | PDP-11 Family and LSI-11 |
| Problem Representation | Equation | Block | Equation | Equation | Equation & Block | Block |
| Data Type | 48-Bit Floating Point | 18-Bit Fixed Point | 48 & 60-Bit Floating Point | 36-60 Bit Floating Point | 32-Bit Floating Point & 16-Bit Fixed Point | 16-Bit Fixed Point |
| Mode of Operation | Interactive | Interactive | Interactive Batch | Batch | Interactive | Interactive |

## APPENDIX B
## DARE BLOCK-OPERATORS USED BY MICRODARE

| | | |
|---|---|---|
| * SUM Z = X + Y | DIV Z = X/Y | |
| * DIF Z = X – Y | * IMULT Z = X * #n | |
| * MULT Z = X * Y | * NEG Z(X | Z = –X |
| * ABSV Z(X | Z = \| X \| | |
| * LIM Z(X | Z = max (X, 0) | |
| COMP Z(X–Y, PL, M1 | | $Z = \begin{cases} PL\ (X-Y \geq 0) \\ M1\ (X-Y < 0) \end{cases}$ |
| SMULT Z(X, CT | | $Z = \begin{cases} -X\ (CT < 0) \\ X\ (CT \geq 0) \end{cases}$ |

| | |
|---|---|
| TRIGG Z(X, Z0 | Schmitt-trigger transfer characteristic |
| * FUNCT Z = Y(X | table-lookup and interpolation in array Y |
| * INTG Z(X * G% | intergrator with integer gain G% |
| SHOLD Z(CT, X | sample hold; Z tracks X if CT > 0 |
| UDELAY Z (Z0, CT, X | unit delay between Z0 and Z; Z0 tracks X if CT > 0 |
| SWEEP Z | sweep waveform (–1 to 1) |
| SAW Z(#n | sawtooth sweep, frequency n |
| PULSE Z (CT, #n, TD | CT > 0 generates n pulses TD units apart |
| NOISE Z(#n, CT, #m | shift-register pseudo-random noise generator |
| STORE AA = X<br>STOGET AA = X | stores time history of X in array AA |
| GET X = AA | produces time history X from array AA |
| TERM X < Y | terminates DRUN when X < Y |
| DISPT X | displays X versus T |
| DISPXY X, Y | displays Y versus X |
| PROB Z(X, RS, WR, CT | amplitude-distribution analyzer |

*Microcode currently developed for Direct Execution Processor.

# APPENDIX C. MICRODARE SAMPLE PROGRAMS

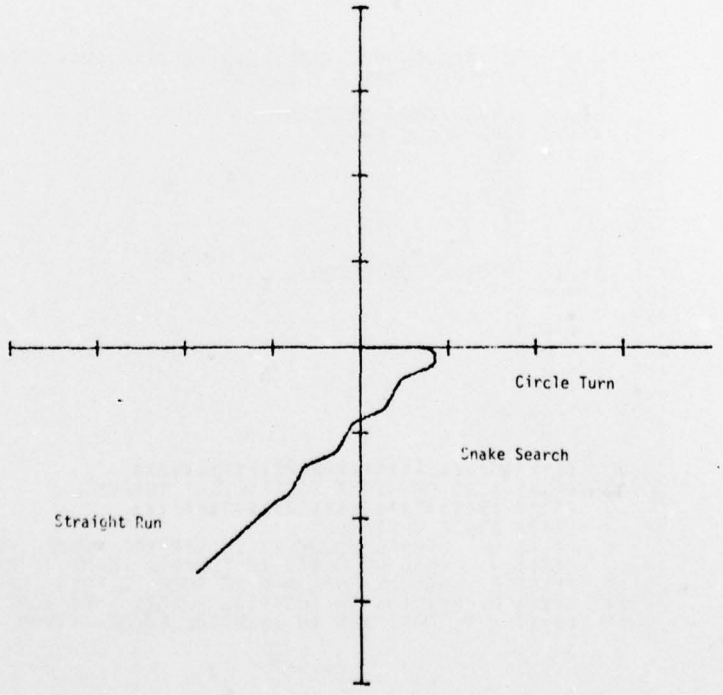
## C.1 Two-Dimensional Torpedo Dynamics Example: by R. H. Hidinger NOSC 6353

```
LIST

10  CLEAR STACK
11  NP = 200: ***** NUMBER OF POINTS
12  DIM GX[NP],GY[NP],QU[NP],QP[NP]
20  DT = 0.00005    ***** TIME INCR TIME SCALE 100<DT=0.005)
30  TM = 0.99    : ***** MAX TIME, TIME SCALE 100 (TMAX = 100 S.)
35  CI% = TM/(NP*DT)
40  ***** RUN CONSTANTS
50  UC = 0.25   : ***** REL. TO 50 M/S
60  DIM RT[129]: ***** RUDDER COMMAND TABLE, FS. = 20 DEG.
65  FOR I = 1 TO 129 : RT = 0.0: NEXT I: ***** ZERO TABLE
70  RT[77] = 0.25: RT[78] = 0.25: RT[79] = 0.25: RT[80] =0.25: ***** CIRCL
E
72  RT[85] = -0.25: RT[90] = 0.25: RT[95] = -0.25: RT[100] = 0.25
73  RT[105] = -0.25: RT[110] = 0.25: ***** SNAKE
76  RT[112] = -0.125: ***** ATTACK
90  ***** DU/DT CONSTANTS
100  X1 = 0.8155
110  X2 = 0.8155
120  UT = X1*UC*UC
130  ***** DY/DT CONSTANTS
140  Y1 = -0.98129
150  Y2 = -0.04561
160  Y3 = 0.12501
170  ***** DR/DT CONSTANTS
180  N1 = -0.96703
190  N2 = -0.44866
200  N3 = -0.77652
210  ***** ROTATION SCALE
220  RS = 0.5
230  ***** SIN AND COSINE TABLES
231  DIM SI[1025],CS[1025]
235  PM = 4*3.1415927: ***** MAX. BEARING, P
240  FOR I = 1 TO 1025

250  A = 2*PM/1024*I-(PM+2*PM/1024): ***** -PM<=A<=PM
260  SI[I] = SIN(A): CS[I] = COS(A)
270  NEXT I
280  ***** INTEGRATION CONSTANTS
300  ***** TIME SCALE = 100
310  QU% = 100
320  QU% = 1600
330  GR% = 10000
340  GP% = 25
350  GX% = 10
360  GY% = 10
400  ***** INITIAL CONDITIONS
410  U = 0.0
420  V = 0.0
430  R = 0.0
440  P = 0.0
450  X = 0.0
460  Y = 0.0
1000  ***** GO!
1010  DRUN
2000  END
20000  ***** ******************************
20100  ***** * 2D CONSTANT COEFFICIENT TORPEDO
20200  ***** ******************************
20210  ***** STATE VARIABLES
20211  ***** U: FORWARD VELOCITY IN TORPEDO FRAME, <50M/S
20212  ***** V: CROSS VELOCITY IN TORPEDO FRAME, < 50M/S
20213  ***** R: ANGULAR RATE AROUND TORP. Z AXIS, <PI RAD/S
20214  ***** P: BEARING IN INERTIAL FRAME, <4PI RAD
20215  ***** X,Y: POSITION IN INERTIAL FRAME, <1000
```

```
20290  ***** **********************************************************
20295  FUNC RD=RT(T
20299  *****
20300  MULT U2=U*U
20400  ***** DU/DT
20500  MULT US=X2*U2                    23950  MULT XD=XC*RS
20600  SUB DU=UT-US                     24000  SUB DX=XB-XD
20700  ***** SOME CROSS PRODUCTS        24010  MULT YA=U*SP
20800  MULT UU=U*U                      24015  MULT YB=YA*RS
20900  MULT RU=R*U                      24020  MULT YC=U*CP
21000  MULT TU=RD*U2                    24025  MULT YD=YC*RS
21050  ***** DV/DT                      24030  SUM DY=YB+YD
21100  MULT T1=Y1*UU                    24100  ***** **** INTEGRATION
21200  MULT T2=Y2*RU                    24200  INTG U(DU*GU%
21300  MULT T3=Y3*TU                    24300  INTG U(DV*GU%
21400  SUM T4=T1+T2                     24400  INTG R(DR*GR%
21500  SUM DU=T4+T3                     24410  INTG P(R*GP%
21600  ***** DR/DT                      24420  INTG X(DX*GX%
21700  MULT T5=N1*UU                    24430  INTG Y(DY*GY%
21800  MULT T6=N2*RU                    24500  *****
21900  MULT T7=N3*TU                    24700  DISPXY X,Y
22000  SUM T8=T5+T6
22100  SUM DR=T8+T7
22150  ***** SIN(P)AND COS(P)
23000  FUNC SP=SI(P
23100  FUNC CP=CS(P
23200  ***** INERTIAL STATE EQNS DX/DT,DY/DT
23500  MULT XA=U*CP
23550  MULT XB=XA*RS
23600  MULT XC=U*SP
```

Two-Dimensional Torpedo Example

```
R HORIZ VERSUS
T VERT

STOP AT LINE 2000
```



C.2 Parameter Optimization Problem: by G. A. Korn, University of Arizona

```
50  CLEAR STACK
55  P=.4
60  DIM FF[33]
65  FOR J%=1 TO 15: FF[J%]=-P: NEXT J%
66  FF[16]=-P/2: FF[18]=P/2
67  FF[17]=0.0
68  FOR J%=18 TO 33: FF[J%]=P: NEXT J%
100  DT=0.01
200  TM=.9
300  GX%=-6
400  GY%=-6
450  G%=1
460  GG%=2
500  A=.7
600  QX=.3
700  QY=.2
1200  Y=0.3
1300  XX=0.999
1400  YY=.999
1500  TX=0.05
1600  TY=0.05
1700  AX=0.1
1800  AY=0.1
1805  N%=160
1810  FOR I%=1 TO N%
1820  X=QX
1830  Y=QY
1840  XX=.9999
1850  YY=.9999
1900  UX=0.0
2100  UY=0.0
2200  VY=0.0
2250  VX=0.0
2300  T=0.0
2400  PRINT I%
10000  ORUN
11000  AX=AX+GG%*UX
12000  AY=AY+GG%*UY
13000  TX=TX-GG%*VX
14000  TY=TY-GG%*VY
15000  NEXT I%
17000  PRINT TX,0.5,TY,0.5*A
19100  END
20100  MULT YD=Y*A
20200  MULT XS=XX*TX
20300  MULT YS=YY*TY
20400  SUM S=X+Y
20410  MULT XT=XX*AX
20420  MULT YT=YY*AY
20500  SUM SS=XT+YT
20600  SUB EE=S-SS
20650  FUNC E=FF(EE)
20700  MULT X1=E*XX
20710  MULT Y1=E*YY
20720  MULT X2=X1*AX
20730  MULT Y2=Y1*AY
20740  MULT X3=X2*T
20750  MULT Y3=Y2*T
22000  INTG X(X*GX%
23000  INTG Y(YD*GY%
24000  INTG XX(XS*0-12
25000  INTG YY(YS*0-12
26000  INTG UX(X1*G%
27000  INTG VX(X3*G%
28000  INTG UY(Y1*G%
29000  INTG VY(Y3*G%
30000  DISPT S
30100  DISPT SS
```

C.3 Second Order Differential Equation/Multiple Runs: by R. Martinez, NOSC Code 6353

```
T HORIZ VERSUS
X,XD VERT


STOP AT LINE 800
>
50  CLEAR STACK
100  TM=0.9999
200  DT=TM/100
300  G%=10
350  Q1%=-10
360  R=.9
370  FOR I%=1 TO 3
400  X=0.8
500  XD=0.0
550  T=0
600  ORUN
710  R=R-0.3
720  NEXT I%
800  END
900  ***** **********
20100  MULT P=XD*R
20300  SUM Q=X+P
20400  INTG XD(Q*G1%
20500  INTG X(XD*G%
20600  DISPT X
20700  DISPT XD
```