A study of portability and retargetability of an ICS

Sharon Beskenis
College of William & Mary - Arts & Sciences

A STUDY OF PORTABILITY AND RETARGETABILITY OF AN ICS

A Thesis

Presented to

The Faculty of the Department of Applied Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Master of Science

by

Sharon O. Beskenis

1980

APPROVAL SHEET

This thesis is submitted in partial fulfillment of

the requirements for the degree of

Master of Science

____

Author

Approved, August 1980

[illegible]

Stefan Feyock

[illegible]

John C. Knight

[illegible]

Robert E. Noonan

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I thank Dr. Stefan Feyock, Dr. John C. Knight, and Mr. Ralph Will for their help, encouragement, and criticism throughout this project. I also thank Dr. Larry Dunning of Old Dominion University for the documentation concerning the installation of I3CS on the DEC 10 computer and Dr. Edwin Foudriat for the information concerning the implementation of the RCA 1802 as a new target machine in I3CS. A special thanks to the Flight Dynamics and Control Division at Langley Research Center and Dr. Edwin Foudriat for making this project possible.

ABSTRACT

The purpose of this study was to design, develop, maintain and document an interpretive computer simulator(ICS) suitable for use by a wide range of research and industrial facilities. The requirements necessary to achieve portability and retargetability were studied and applied to the design and development of an Interactive, Instrumented ICS, I3CS. This system was developed on the CDC NOS operating system.

The NASA Standard Spacecraft Computer 1 (NSSC-I) was used as a target machine in one application of I3CS. It has been rehosted and checked out on another machine, Old Dominion University's DEC 10, and has had the simulator portion retargeted in-house to the RCA 1802 in another instance. This system has proven to be portable, retargetable and useful.

INTRODUCTION

The flexibility of design and the relatively low cost of combining integrated circuits into a system have encouraged the development of a multitude of microprocessors to meet the growing needs of industry. Advances in microprocessor technology have enabled engineers to fashion systems tailored to their specific needs which may vary from flight computers to industrial robots to monitors of various systems or processes. Since an increasing number of these microcomputers is becoming available, the range of sizes, architectures and instruction sets is becoming overwhelming.

Determining which processor is best suited for a particular application is a difficult task. Certainly purchasing, implementing and benchmarking several microprocessors to select the best one is not the answer. A more feasible solution might be to purchase or construct simulators for each system. Costs are still high, however, although better results can be obtained in less time by this method.

A portable, retargetable, interpretive computer simulator(ICS) seems to be the most logical approach. The proposed simulator, which is written in a high level language, resides on a host computer and interpretively executes the machine code of a given target computer, the microprocessor being analyzed. The execution of the microprocessor is simulated at the bit level. This means, for example, that one's or two's complement arithmetic is performed depending on the microprocessor being studied, regardless of how the host computer operates. The ICS includes a clock, an interrupt mechanism and I/O devices which are all implemented in this high level language. Only specialized functions such as Boolean operations may be simulated in the host machine's assembly code for greater speed, if desired, but will have to be rewritten if the ICS is ported to a new host machine.

Instrumentation, software probes into a program, is an enhancement to an ICS. This capability not only aids in processor selection and benchmarking but also is an important feature for program development and debugging once a microcomputer has been selected. Timing information, checking for invalid values, and detecting the execution of incorrect paths can be made possible with instrumentation. Memory and register values can also be displayed at various points in the program execution. An interactive, instrumented, interpretive computer simulator is most attractive since the user can be given complete control over the simulation and the ability to make several runs in one interactive session for a given target machine exists. These features combined decrease the amount of time needed to obtain various statistics from the simulator and to debug programs.

Most interpretive computer simulators have the drawback that they can only be executed on a particular machine or operating system. A portable ICS is clearly much more useful and desirable. Not all sites within an organization have the same machines or operating systems, but the need to have the capability to use the ICS on all of their equipment often exists. Unfortunately, portability is a characteristic that is often overlooked when designing an ICS as was the case with Reference 1.

Another important and desirable characteristic is retargetability. Most previous ICS systems have been targeted to a particular machine or a particular class of machines such as the software simulator of the Minuteman D17B computer [1] or CDC's simulator of the Intel 4004, 4040, 8008, 8080 series machines [2]. If an ICS is written in a generalized manner for retargetability, a great savings of time is realized because a minimal amount of effort is required to simulate other microprocessors. Hopefully, it will no longer be necessary to purchase an ICS for each microcomputer under consideration by an organization since the ICS will have proved to be retargetable.

CHAPTER 1

DESIGN CONSIDERATIONS

1.1 Design Overview

Several factors were considered in the design of an ICS system. These considerations include

1. portability
2. modularity
3. retargetability
4. instrumentation
5. an interactive capability

Portability, the ability to rehost the interpretive computer simulator system on a wide variety of operating systems, and retargetability, the ability to substitute alternate machines with vastly different architectures as the target machine, are the most important design considerations. Other advantageous features include instrumentation, a feature that aids in program development, debugging and benchmarking, and interactive execution of the ICS for greater user control over the simulation. Since a number of interactive and/or instrumented interpretive computer simulators are currently available, concentration on the important aspects of designing a portable, retargetable ICS will be made.

1.2 Designing for Portability

In order to achieve portability, the ICS must be written in a programming language whose compiler is either resident or can be installed on the computers of interest. High level languages meet this criterion. The language choice is very important because it can enhance portability. Some languages are obviously better suited for a particular application than others and make porting from one computer to another much simpler.

Fortran is the language that most often comes to mind due to its availablity on most operating systems and its standardization. Languages similar to Pascal, however, are better suited for software support tools for the following reasons. Pascal is a very powerful language that has the facilities to handle user-defined data structures, linked lists and bit manipulation, all of which are necessary for an ICS. Since Fortran lacks these facilities, a great amount of effort is required to accomplish the task and a lack of clarity too often results. Pascal data structures provide the capability for clear and concise code and thereby enhance portability due to the ease with which necessary changes can be recognised and made. Pascal lends itself to structured programming which tends to cut down on program errors, aiding in the production of portable programs.

1.3 Arithmetic Problems Affecting Portability

An obstacle to be overcome when designing a portable program is the problem of performing target machine arithmetic operations in one's or two's complement regardless of the host machine's method of arithmetic. If changes in this area can be minimized, portability can be maintained without too much difficulty. There are four possible combinations to be considered: one's complement host - one's complement target, two's complement host - two's complement target, one's complement host - two's complement target, and two's complement host - one's complement target. Naturally, the first two combinations present the least number of problems. Arriving at a design to take care of as many of the cases as possible is necessary.

The following proposed design resolves this problem in most cases. Arithmetic registers are defined as integer fields which have a fill field and a data field as subfields. A sample declaration for an 18-bit register on a 60-bit host follows to illustrate this concept.

```
REGISTER = PACKED RECORD
             CASE INTEGER OF
               0 : (FILL : BIT42;
                    DATA : BIT18);
               1 : (INT  : INTEGER);
           END;
```

One's and two's complement number representations are the same in the positive range. Only non-positive numbers differ in representation. By loading the data fields with two's complement numbers and the fill fields with zeroes, operations between REGISTER.INT and another REGISTER.INT can be performed on a one's complement machine with no adjustments necessary. This is due to the fact that the negative two's complement numbers in the 18-bit data field of the example look like positive one's complement numbers when the entire 60-bit integer field is used. Since there is no difference in operations on positive one's or two's complement numbers, the REGISTER.INT field can be used for addition or subtraction and the REGISTER.DATA field can be peeled off for the correct two's complement result.

An illustration of two's complement addition on a one's complement host of 3+(-3), 3+(-2), and -3+(-1) follows. The host machine word size is 30 bits and the target machine word size is 18 bits with 12 bits of filler for this example. Octal representation of the numbers is used.

```
  3+(-3)=0          3+(-2)=1          -3+(-1)=-4

0000|000003       0000|000003       0000|777775
0000|777775       0000|777776       0000|777777
-----------       -----------       -----------
0001|000000       0001|000001       0001|777774
     ------            ------            ------
```

The correct two's complement results are contained within the 18-bit data field. Had only the 18-bit data field been used for the additions instead of the entire 30-bit word, erroneous 2s complement results would have ensued. The addition of negative 18-bit numbers would have 000001, 000002, and 777775 as the respective results. When adding one's complement numbers on a two's complement host, the fill fields are zeroed out, the data fields are loaded with the one's complement numbers, the addition is performed on the entire host machine word, and the carry into the fill field of the result is added to the resulting data field for the correct answer.

It is possible that the size of the target machine's registers exceeds the host machine's word size. An adaptation of the previously mentioned register design suffices in this case. Suppose, for example, that the host machine had a 32-bit word and the target machine had a 60-bit register size. Two host machine words would be required to handle the target machine register. A sample register definition for this case follows.

```
REGISTER = PACKED RECORD
             CASE INTEGER OF
               0 : (Fill1 : BIT2;
                    DATA1 : BIT30;
                    FILL2 : BIT2;
                    DATA2 : BIT30);
               1 : (INT   : ARRAY 1..2  OF INTEGER);
             END;
```

Operations between REGISTER.INT 1 , REGISTER.INT 2 and another REGISTER.INT 1 , REGISTER.INT 2 will be correct when the fill fields are set to zero because the addition or subtraction will appear to be an operation between positive numbers. The result in the REGISTER.FILL2 field must be added to REGISTER.INT 1 in order to reflect any carry out of the addition or subtraction of the lower half of the registers. REGISTER.DATA1 and REGISTER.DATA2 will contain the correct 60-bit result.

1.4 Modular Design

A program that has been broken up into procedures according to function will be called a modularized program. Using a top-down structured design, one module may be a loader and another module may be a simulator. Each of these modules is further refined by being broken up into smaller pieces according to function such as a fetch module and an execute module within the simulator module, and so on, until the pieces are quite small, the functions are quite specific, and the dependence upon other modules is ideally non-existent. Modification of a modularized program is much easier than altering one that is not modularized because typical straight-line code is extremely difficult to follow unless the program is very small. A program broken up into modules, each of which has a specific function, is easy to understand and alter, if necessary, aiding portability. Modularization also aids in the development of retargetable programs. Due to the functionality of each module, it is easy to select which modules should be changed or replaced when retargeting to a new microprocessor.

1.5 PDL as a Design Tool

Program Design Language(PDL) by Caine, Farber and Gordon, Inc. [3] was chosen to design and document the framework for I3CS. PDL is designed for the production of top-down structured designs. By using "structured English", a complete design which contains all external and internal interface definitions, identification of all procedure calls, global data definitions, control block definitions and specifications of all the processing algorithms of all procedures can be produced by PDL before any code is written. This working document not only is self-documenting by virtue of the fact that the English language is used in the design but also facilitates keeping data consistent from module to module by means of data and segment(procedure) indexes [4]. The flow is easily determined by studying the reference tree which shows how segments are nested. This valuable design tool simplified the design of a portable and retargetable system due to the clarity of the working document. Although the PDL document was not updated to reflect the final version of the ICS, coding from the initial PDL working document into Pascal structures for the baseline ICS was not difficult.

A PDL version of the main program for I3CS follows. The ref pages refer to the pages within the PDL document on which the other modules are defined. For example, the Initialize System Variables module is defined on page 8 of the document. The DO-ENDO represents the flow of the segment or module execution. This flow is repeatedly executed until Stopflag is true. The equivalent Pascal code follows the PDL example.

PDL Main Program Definition

```
 ref
page ********************************************
   8 * Initialize System Variables              *
     * DO Repeat Until Stopflag                 *
  12 *      Read and Set Simulator Memory       *
  17 *      Read and Set Test Instructions      *
  42 *      Machine Execution                   *
     *      Process Data Output                 *
     * ENDO                                     *
     ********************************************
```

Pascal Main Program

```
Begin
  Initializer;
  Repeat
    Loader;
    Syntaxanalyser;
    Simulator;
    Outputformatter;
  Until Stopflag;
End.
```

1.6 Designing for Retargetability

The capability for retargeting to another mini or microcomputer was an essential characteristic necessary for developing a versatile I3CS system. The following design rationale was used to achieve retargetability. One host machine word of memory should include the following information:

1. a filler area
2. the instrumentation bits
3. the target machine word of memory

For example, if the host machine has a 32-bit word size, the instrumentation requires 7 bits, and the target machine has an 18-bit word, one host machine word of simulator memory will contain 7 filler bits, 7 instrumentation bits and 18 bits of actual target machine memory.

Since it is possible for the target machine's word size to exceed the host machine's word size, a method of handling this case without sacrificing retargetability is needed. If the target machine had a 60-bit word size and the host machine had a 32-bit word, for example, three host machine words would be required to handle one target machine word of memory. The first word would contain 25 filler bits and 7 instrumentation bits. The second and third words would each contain 2 filler bits and the upper and lower 30 bits of target machine memory, respectively.

Any changes in the size of either memory or registers, in the number of registers or memory words, or in the instrumentation can be easily accomplished when retargeting if all of this information is readily accessible. One means of assuring the accessibility of data is to have the information be global data. These changes can be made by merely filling in the appropriate blanks in the memory and register definitions. Additional registers or memory layout fields can be added easily within the framework of these definitions.

A partial example of the changes necessary when retargeting from the NSSC-I to the Nova 1200 follows. Note that both the fill-in-the-blanks strategy and the additional-field strategy were used. The FILL and DATA fields were changed by filling in new values: BIT7 versus BIT5 and BIT16 versus BIT18. Additional fields of target machine memory were added, the 16-bit Nova fields 1 and 2, and the NSSC-I fields 1, 2, 3, and 4 were deleted. Only the previously mentioned field 0 remains.

NSSC-I Memory Formats

```
MEMORY = PACKED RECORD
   CASE INTEGER OF
     0 : (FILL : BIT5;
          INST : BIT7;
          CASE INTEGER OF
             0 : (DATA      : BIT18);
             1 : (IX        : BIT1;
                  FILL1     : BIT1;
                  VAL       : BIT16);
             2 : (MAJOROP   : BIT5;
                  MAJORIX   : BIT1;
                  MAJORADR  : BIT12);
             3 : (MINORFIL  : BIT12;
                  MINOROP   : BIT6);
             4 : (SIGN      : BIT1;
                  MAG       : BIT17));
     1 : (INT  : BIT30);
         END;
```

Nova 1200 Memory Formats

```
MEMORY = PACKED RECORD
   CASE INTEGER OF
     0 : (FILL : BIT7;
          INST : BIT7;
          CASE INTEGER OF
             0 : (DATA      : BIT16);
             1 : (INDICATOR : BIT1;
                  CASE INTEGER OF
                    0 : (ACC    : BIT2;
                         IADDR  : BIT1;
                         IX     : BIT2;
                         DISPL  : BIT8);
                    1 : (ACCIO  : BIT2;
                         TRANS  : BIT3;
                         CNTRL  : BIT2;
                         DEVICE : BIT6);
             2 : (ACS       : BIT2;
                  ACD       : BIT2;
                  OPAL      : BIT3;
                  SHIFT     : BIT2;
                  CARRY     : BIT2;
                  NOLOAD    : BIT1;
                  SKIP      : BIT3));
     1 : (INT   : BIT30);
         END;
```

The simulator should be a complete module, due to its functionality and the fact that most changes when porting or retargeting I3CS should be isolated to this area. Since target machine memory, the instrumentation and the registers are global, all changes necessary to retarget from one machine to another should be essentially isolated to the simulator itself. Much of the simulator can remain intact under portation when the functions within the simulator are modularized. The framework remains basically the same:

1. execute an instruction

2. process "run-time" requests (time out, time dump, interrupt, and I/O requests)

3. satisfy instumentation requests

4. increment the program counter

Procedures to perform specific functions such as right shift can be utilized by any target machine. The interrupt and I/O request handler may also be suitable for the most part for many different target machines. The actual execution of specific target machine instructions will differ, however, since the operation codes differ from one target machine to another and different flags may be raised during execution. By isolating most changes to the global data and the simulator module, retargetability appears to be an attainable goal.

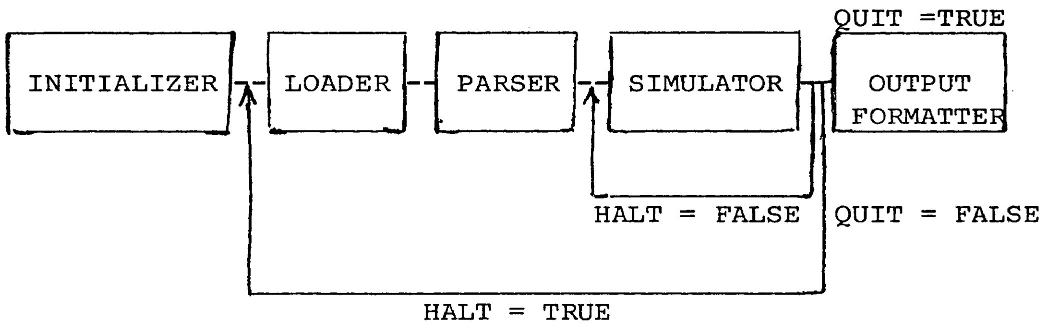
CHAPTER 2

SYSTEM OVERVIEW

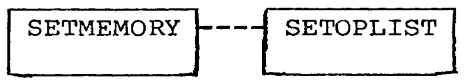
I3CS(Interactive, Instrumented, Interpretive Computer Simulator) is broken up into five major modules to provide a portable and easily retargetable package. These primary modules are the initializer, the loader, the syntax analyser, the simulator, and the output formatter. These modules work together within I3CS in the following manner. The initializer module is invoked to set target machine memory, registers and other values to zero. Next, the loader module is invoked to load target machine data and instructions into the simulator memory. The initial program counter value is also loaded. Simulated "run-time" events such as interrupts and I/O requests as well as two time-oriented commands, time out and time dump, will be referred to as "run-time" or operating commands. Any operating requests on a user-supplied text file, the operating file, are processed in this module. The parser module is then invoked. Commands supplied interactively by the user to set memory, registers and instrumentation or to start or stop the simulation are processed. The simulator module is invoked next to execute instructions in the target machine memory and advance the program counter. This process continues with instrumentation requests and operating commands being satisfied until a halt flag or a quit flag is set.

Upon termination of the simulation process, one of two paths is taken based on the value of the boolean variables, halt flag and quit flag. The halt flag halts the simulation and returns control to the user whereas the quit flag terminates the entire simulation process. If the halt flag is true, the system loops back through the loader, the syntax analyser and the simulator modules. This enables the user to supply any combination of the following: new operating commands; new instrumentation requests; or either new data using the same set of instructions, a new program using the same data, or a totally new set of data and instructions. If the quit flag is true, the output formatter is invoked to print information requested by the user. This module may be written in Pascal by the user in order to print the information supplied by instrumentation and operating list commands in a desirable format. An output formatter is supplied, however, for those who do not care to write their own version. Block diagrams of I3CS and of the loader, the parser, and the simulator modules follow.

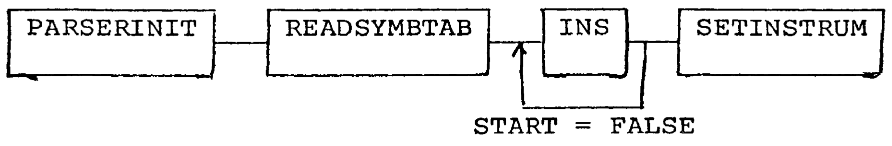
Block Diagram of I3CS



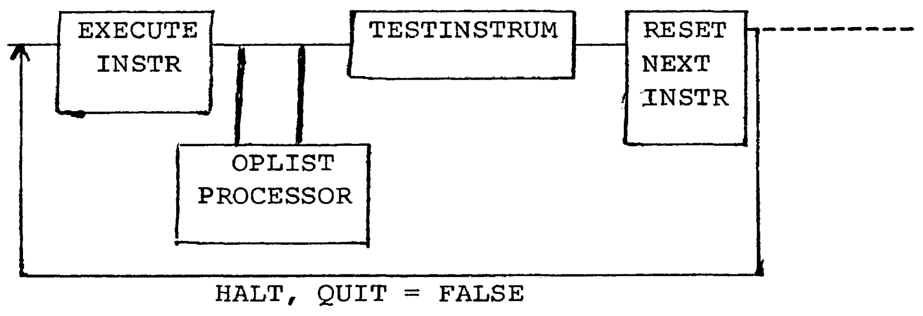
Loader Diagram



Parser Diagram



Simulator Diagram



Because each module in I3CS has a specific function, portability and retargetability are enhanced. The loader and the syntax analyser remain unscathed when retargeting. Except for a few minor possible changes, the initializer can also remain intact. Changes to the system are for the most part isolated to the simulator module when retargeting. It was intended that the output formatter be user-defined in order that individual preferences in the instrumentation information display could be accomodated. The RCA 1802 was substituted as a new target machine in I3CS with relative ease. Problems encountered when porting from one operating system to another can be easily isolated and reconciled due to I3CS's modular design. Old Dominion University has installed this system on their DEC 10 computer. It is for these reasons, portability and retargetability, that I3CS is different from other previous ICSs.

CHAPTER 3

EVOLUTION OF I3CS

3.1 Nova 1200 Simulator

The original ICS used the Nova 1200 as the target machine. This implementation was used primarily as a learning tool prior to the development of a more sophisticated interpretive computer simulator system. This system was not interactive and it had no real loader. Simulator memory was read in from a file and loaded contiguously from location 0 to location N and the instrumentation was "hard-wired" in the initializer. An output formatter was called after the execution of each instruction in order to display trace, checkpoint, and memory protect information. This system took 3 man-months to develop. 5.251 CP seconds were required to compile the

ICS and 20000 octal words of memory were required to execute.

Several things were learned from this primitive system. Basically this was an exercise in using Pascal data structures and in realizing how powerful those structures could be if used properly. Although a fill-in-the-blanks philosophy was used for memory and register definitions, it soon became obvious that these structures could become more streamlined by using Pascal more effectively. This change would make the notion of easy retargetability a reality. Undue hardship was placed on the user by expecting him to essentially "hard-wire" instrumentation information. Development of a command language to accomplish this task, as done in other ICS systems, appeared to be a more logical approach. It would be necessary to design the command language processor so that it in no way interfered with the portability and retargetabilty of the finished product. This exercise also provided some insight into the requirements for a loader such as the ability to load blocks of memory which are not necessarily contiguous.

3.2 I3CS System

3.2.1 Introduction

I3CS, the final product, incorporated all the lessons learned from the Nova 1200 ICS. As mentioned earlier, I3CS consists of five major modules. Since the initializer is trivial and the output formatter is meant to be user-supplied, the loader, the syntax analyser, and the simulator will be discussed in terms of their functions and how they meet the requirements of portability and retargetability.

3.2.2 Loader Module

The loader module consists of two basic procedures: the loader and the operating list processor. When the user writes an assembly language program, he must assemble it and feed the load file through a user-written interface program in order to create MACHFILE, a load file formatted for I3CS. The loader loads target machine memory from MACHFILE, sets the program counter, and reads in a program name to be associated with the run. A code heading on each MACHFILE record enables the loader to determine whether a program name, an instruction counter or a block of simulator memory is to be read in. The number of characters necessary to make up one word of simulator memory is determined by I3CS given the word size and the base of the target machine. The characters making up one word, in octal or hexadecimal representation, are then converted to an integer value. Blocks of memory, which need not be adjacent to one another, are read in at one time. The first word of each block contains the starting address of the block into which the particular block of information is to be loaded. The remaining words are data which is read and loaded, one at a time, into the memory block until an end-of-line is encountered. Now either another block of data, a program counter value or a program file can be read in until the end-of-file is encountered. No changes are required when retargeting due to the fact that the loader automatically computes how to read in the file based on the target machine specification of the word size and the base.

The operating commands processor reads in operating commands from the operating file, OPFILE, which is a text file built manually by the user. These commands consist of interrupt, cycle steal I/O, time dump, and time out requests. The commands are inserted onto an operating list, OPLIST, in order by time for use by the simulator module. TIMEOUT 105.0 which sets the halt flag when clock time equals or exceeds 105.0 ms and INTERPT 2 10 152.0 162.0

which causes an interrupt on channel 2 to occur when clock time is greater than or equal to 152.0 ms and less than or equal to 162.0 ms are sample operating commands. Further examples of these commands are contained in Appendices B and D. Portability and retargetability are not hindered by the operating list handler. Operating commands are independent of target machine.

3.2.3 Syntax Analysis Module

The syntax analyser performs many tasks. It reads in the symbol table information which is in alphabetical order by symbol name from the symbol table file, SYMFIL. When an assembly language program is assembled, a load file containing loader and symbol table information is fed through a user-written interface program that formats the symbol table for SYMFIL. This information is stored into an array of symbol records containing the label, the location associated with the symbol, the program name and the section number. This enables users to refer to locations by a label plus an offset which may be positive or negative. It is convenient for the user to designate locations in this way rather than to compute the actual address. When a label is used in a command, a binary search of the symbol table array on symbol name is performed until a match is found. The address associated with that symbol plus the offset is the address to be used in the instrumentation. If the symbol is not in the symbol table, however, an error message to that effect is emitted and the user is given another chance to specify the command with the proper location.

The actual parser portion of this module contains a scanner, syntax analysis procedures and error handling. Commands are read into an input buffer and lexical analysis is performed to discover tokens. The scanner places characters into an "identifier" array, ID, until it reaches a separator. One character lookahead strategy is used. Separators include ",", "'", "(", ")", ".", "=" as well as the operators: "+", "-", "*", "/". Multiplication and division have precedence over addition and subtraction and the command language is left associative.

The syntax analysis of this language was implemented using recursive descent parsing. A set of recursive procedures are used to recognize the input with no backtracking necessary. An adaptation of the reserved word strategy is used whereby keywords are treated as identifiers. A table of reserved words is checked to see if the identifier is a keyword [5]. Based on the keyword and the separators obtained while parsing the command, the syntax analyzer is directed accordingly.

In order to successfully use recursive descent pasrsing, the parser must be able to tell, given the current input symbol a and the nonterminal A to be expanded, which one of the alternates of the production $A \rightarrow a1 | a2 | \ldots | an$ is the unique alternate that derives a string beginning with a. The proper alternate must be detectable by looking only at the first symbol it derives [6]. This language is simple enough that the keywords and separators encountered enable the parser to always choose the correct alternate.

If a command is not formulated properly, the syntax analyzer recognizes it and emits error messages accordingly. Parsing does not halt, however. Once a line of input has been parsed, correct commands are accepted, errors are flagged and control is turned over to the user who can then type in new commands and correct and resubmit any erroneous commands. Error recovery of this type is very important if I3CS is to be a successful interactive program. A list of error messages and a sample command language input containing seeded errors are available for inspection in Appendix C.

As commands are parsed, linked lists containing set memory, set registers, and instrumentation requests are built. This list is processed after all the commands have been input by the user and the appropriate registers or memory bits are altered. The TESTVAL command appears on this list and either sets the test instrumentation bit of a particular memory location to "on" or "off". When this bit is set to "on", the upper bound and the lower bound associated with that particular location are placed onto a test list by location. If a memory location whose TESTVAL bit is set is reached during the simulated execution of a program, a search of the test list is made for an entry associated with that particular location. The contents of memory at that location are checked to see if the value is between the upper and lower limits specified by the test list entry. Likewise, when this bit is turned "off", the test list is searched until that location is found and the entry containing the location and its bounds is removed from test list. TDUMP and TIMEOUT requests are inserted onto the operating list, OPLIST, according to time as in the loader module.

This module is designed for retargetability because the symbol table handler, the command language processor, and the instrumentation can be accomplished with no regard to the target machine being used. Only changes to the way the interactive command language file is handled may affect portability because there is no standardized means of handling interactive I/O. This module is quite portable, however, because the changes needed to alter the interactive I/O problem are easy to spot. They are confined to the scanner.

3.2.4 Simulator Module

The simulator module operates in the following manner. An instruction is fetched and executed, an attempt to satisfy operating list requests is made and instrumentation requests are satisfied. This cycle continues until the halt flag or the quit flag has been set. When the halt flag is set, the simulation stops and control is returned to the user who can then issue commands requesting additional information and either resume or stop the entire simulation process. The halt flag can either be set by the program being interpretively executed via a halt instruction or by the user by setting halt bits in memory. The quit flag, when set, causes the simulation process to come to a complete halt. The user must issue the quit command in order to terminate execution of I3CS.

Once an instruction is fetched and executed, the instrumentation bits associated with that instruction are tested. These instrumentation requests, if any exist, are then satisfied. Appendix C, the command language writeup contains a discussion of the instrumentation functions.

If the operating list, OPLIST, is not empty, one operating list request may be serviced between execution cycles if it meets the time constraints. Therefore, after an instruction executes, if OPLIST is not empty, the first item on OPLIST, the item having the lowest starting time, is selected for servicing. This request is satisfied if the starting time is less then or equal to the current clock time unless it is an interrupt or an I/O service request. If the request meets this criterion, then it is serviced and removed from OPLIST; otherwise, it remains on OPLIST to be serviced at a later point in time. In the case of interrupts or I/O service requests, the request is satisfied if the starting time is less than or equal to the clock time and the clock time is less than or equal to the ending time for the request. If the criterion is met, it is serviced and removed from OPLIST, otherwise, the next item from OPLIST is selected for possible service.

It was intended that all major changes necessary to retarget I3CS would be performed within this module. Fetch the next instruction, process operating list commands, and perform certain specialized tasks within the execution phase such as right shift should remain basically intact. The execution of the target machine instructions will be different and the information desired when instrumentation requests exist may have to be reformatted somewhat. Such changes are to be expected when retargeting and can be made within the general framework of this module with little difficulty.

CHAPTER 4

IMPLEMENTATION OF THE NSSC-I IN I3CS

4.1 NSSC-I Characteristics

The NSSC-I(NASA Standard Spacecraft Computer 1) flight computer was chosen for simulation. Not only is this computer widely used but also the instruction set is broad enough for development of a reasonable simulator. The NSSC-I has an 18-bit word, two's complement arithmentic, three fixed point registers: the accumulator; the extended accumulator; and the index register and two major status registers: the lockout status register and the storage limitation register. Two types of I/O, cycle steal and program controlled I/O, are also available on the NSSC-I making it an even more appealing choice as the target machine.

The NSSC-I has two instruction formats: memory access(major opcode) and non-memory access(minor opcode). These formats are depicted in Appendix A. The lower order 12 bits of a major opcode instruction limit an instruction to 4096 memory locations when using direct addressing. The NSSC-I has the ability to address 16 memory banks of 4096 locations by combining a 4-bit page register with the 12-bit operand address to form a 16-bit address. The zero page was always used in I3CS in order to limit the amount of memory required by the simulator. By increasing memory size, more than one page can be addressed because the paging mechanism was built into the simulator.

Effective addresses can be altered in two ways during execution, by indexing or by using indirect instructions. If the index bit in a memory access instruction is set, the contents of the index register are added to the operand address with the sum becoming the effective address. The contents of the location specified becomes the effective address when an indirect addressing instruction is executed. If indexing occurs when an indirect addressing instruction executes, the contents of the location specified in the operand address plus the index register contents becomes the new effective address.

The storage limitation register(SLR) is used to specify a protected block of memory. Bits 0 - 8 of the SLR specify the lower limit(L) and bits 9 - 17 specify the upper limit(U). If the effective address is within the limits, $L \leq EA \leq U$, then a write into EA is permitted. All of the aforementioned characteristics of the NSSC-I made it an interesting choice for the I3CS simulator.

4.2 NSSC-I Implementation Statistics

I3CS required seven man-months to develop. The implementation and testing of the NSSC-I as the target machine required an additional 1.5 man-months. Software sneak circuits were introduced to ensure that overflows and carries, for example, were detected and treated properly. Test cases written for NASA Goddard to check out NSSC-I simulators were run successfully through I3CS. Interrupts and storage of data into valid and invalid areas were tested extensively as were all of the instructions.

I3CS requires 27.121 octal CP seconds to compile using the NSSC-I as the target machine. 42241 octal locations are required to load and execute. This figure includes 21000 octal locations for the heap which is needed when many instrumentation and operating list requests are made. An experiment was conducted using the segmented loader to cut down on the memory required to load and execute given the heap size of 21000 octal locations. This resulted in a field length of 34000 octal locations necessary. By further segmentation, the amount of field length will be reduced even further but a sacrifice must be made. More time to execute will be required in order to get the segments swapped in and out of memory. Appendix D contains a sample run of I3CS with the NSSC-I as the target machine.

CHAPTER 5

INSTALLATION OF I3CS AT ODU

5.1 Introduction

As has been stressed earlier throughout this presentation, portability was a very desirable design requirement for a more useful and versatile interpretive computer simulator. I3CS was ported to Old Dominion University's DEC 10 which has a 36-bit word and performs two's complement arithmetic. It was very interesting to see just how portable I3CS was since it was developed on a one's complement host.

5.2 Modifications Required by the DEC 10

A few changes were necessary in order to port I3CS to the DEC 10 computer. One expected modification was the adjustment of fill fields in memory and register definitions and of the upper bounds of the range specifications for some variables. These alterations reflect the change from a 60-bit host to a 36-bit host machine word size. Other minor alterations included the following:

1. Change $ to " for hexadecimal constants.
2. Change the definition of the variable, HEXDIGIT, from a set of 'A'..'9' where 'A'..'F','0'..'9' is the active set on NASA Langley Research Center's version of CDC NOS to a set of '0'..'F' for Old Dominion University's system. This reflects a change in the character set collating sequence.
3. Change reads, writes, and formal parameter passing of packed arrays to unpacked arrays.
4. Cease the packing of negative numbers(used to denote signed integers). An example of this is the following: Sgbit6 = -"1F.."1F.

More extensive changes included the heap management, set handling, and the file system. Each of these topics are treated in the following paragraphs.

There is no garbage collection in Old Dominion University's version of Pascal. The dispose command has no effect on their DEC 10 system. The heap management commands, new and dispose, are simulated by keeping a free space list for each pointer type. A dispose procedure adds an item from the heap to the appropriate free space list. A new procedure checks the appropriate type free space list for any pointer elements. If this list is empty, a Pascal new command is executed, otherwise, an element is delinked from the free space list and used.

Set handling is different on the DEC 10. Sets take two words or 72 objects. By switching sets to packed arrays of Boolean, a true value indicates that a subscript is in the set. By accessing a packed array of Boolean and storing it into a variable that is overlayed as a set and a packed array of Boolean, set operations can be performed.

The DEC 10 Pascal RTS(Run Time System) has no segmented files and has a truly interactive file system. All references and operations for segmented files can be replaced with standard file references. A small driver program was written to ask what files will or will not be used. A boolean value telling whether or not a particular file is used determines if a reset or rewrite should be performed. A reset or rewrite requires assignment of a value to the buffer variable which is undesirable when the file is not to be used.

5.3 Implementation Statistics

The utility package SOUP, a file comparison program resident on Old Dominion's system, was used to flag changes to the baseline CDC version of I3CS. It is for this reason that a very accurate description of all of the changes made when porting I3CS to the DEC 10 system exists. Two man-months were required to port this system to the DEC 10. This figure includes the time spent learning how to use utilities such as SOUP and the text editor, tools to assist in making and documenting all necessary changes, and how the CDC and the DEC 10 versions of the Pascal compiler differ. Dr. Larry Dunning of Old Dominion University estimates that a similar task would now only take a couple of man-weeks since there would no longer be a learning curve.

CHAPTER 6

RETARGETING I3CS TO THE RCA 1802

6.1 RCA 1802 Characteristics

The RCA 1802, which performs two's complement arithmetic, was chosen as the new target machine for I3CS. The RCA 1802 architecture is based on a register array comprising sixteen general-purpose 16-bit scratch-pad registers. The high-order or the low-order byte of a particular scratch-pad register, R, may be referenced. Three 4-bit registers labeled N, P, and X hold the hex digits used to select individual scratch-pad registers in the following manner: R(N), R(P), or R(X). Register P contains the hex digit value determining which of the registers is being used as the program counter. The unique capability to specify any one of 16 registers as program counter in a single instruction is provided. This feature makes it possible to maintain pointers to several different programs simultaneously and to transfer control quickly from one to another. A pointer to a program that services interrupt requests is a special and important example of this feature. The I register is used to determine the instruction type.

The RCA 1802 has three instruction formats: one-byte, two-byte, and three-byte. These formats are depicted in Appendix B. Most instructions are the one-byte type. The first machine cycle fetches or reads the appropriate instruction byte from memory and stores two hexadecimal instruction digits in registers I and N, respectively. The operation specified by I and N is performed in the second machine cycle. I specifies the instruction type and N either designates a scratch-pad register or acts as a special code. Immediate and short-branch instructions have a two-byte format. The first byte contains values for I and N and the second byte contains either an operand or an address. Long-branch instructions and long-skip instructions have a three-byte format and require one fetch and two execute cycles. I and N are specified in the first byte. The high-order and low-order address bits are contained in bytes two and three for the long-branch instructions. Bytes two and three contain the first and second skipped byte for long-skip instructions.

There are four basic addressing modes in the RCA 1802 architecture: register, register-indirect, immediate, and stack. In register addressing, the address of the operand is contained in the N-field of the instruction byte. Direct addressing of any of the 16 scratch-pad registers for the purpose of counting or moving data in or out of registers is possible. In register-indirect addressing, register N specifies one of 16 scratch-pad registers whose contents are the address of the data in memory. R(P) addresses memory so that the operand is the byte following the instruction in immediate addressing. In stack addressing, one specific CPU register is implied as the pointer to memory. Usually R(X) is used. By using stacks for working space, immediate addressing for all constants, register pointers for tabular and vector arrays and the registers themselves for miscellaneous counters and switches, optimal use of program space is made.

6.2 Modifications Required to Retarget I3CS

Modifications to the simulator and the output formatter, which was intended to be user-supplied, were required when implementing the RCA 1802 as a new target machine in I3CS. The initializer, the loader, and the parser modules remained intact, however. The new simulator module was built within the same framework as the NSSC-I simulator. Modules performing the execution, satisfying instrumentation requests, and performing operating command execution, if time constraints were met, still existed. The appropriate changes were made to reflect the new target machine's architecture. Specific details within the simulator to reflect the way interrupts are handled, the new target machine's instruction set, and the setting of various registers when certain operations are performed, for example, had to be reprogrammed.

At the memory and register definition level, changes were made to define the new memory and register formats. New register definitions were added using the previously described additional-field strategy and a fill-in-the-blanks strategy was also used. Memory definitions were also altered by these two strategies. These modifications required 2.5 man-weeks to complete. A successful simulation of the RCA 1802 was attained.

CHAPTER 7

CONCLUSIONS

I3CS was designed for portability and retargetability. Whether or not an interpretive computer simulator containing both of these features could be written was questionable. The ease with which the system could be ported and retargeted was another major concern. I3CS has proven to be both portable and retargetable. It resides on the both the DEC 10 ana the CDC Cyber Series computers using the NSSC-I as the target machine. I3CS was retargeted to the RCA 1802 using the CDC Cybers as hosts. A minimal amount of effort was required for retargeting and porting this system to another host. Having an ICS that is both portable and retargetable has proven to be not only feasible but also easily attainable.

APPENDIX A

NSSC-I Instruction Formats

There are 55 instructions, 31 of which require memory access and are referred to as major opcodes. The other 24, referred to as minor opcodes, have an opcode in the operand field of the instruction word. These instructions do not access memory. An illustration of the two formats follows.

MAJOR OPCODE - MEMORY ACCESS

B 17	B 16	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1	B 0
OPERATION					IN-DEX	OPERAND ADDRESS											

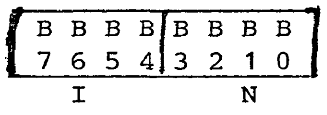
MINOR OPCODE - NON-MEMORY ACCESS

B 17	B 16	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1	B 0
0	0	0	0	0	NOT USED							OPERATION					

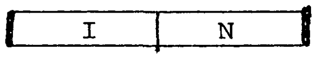
APPENDIX B

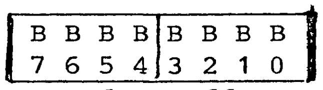
RCA 1802 Instruction Formats

One-Byte Instruction



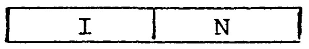
Two-Byte Instruction

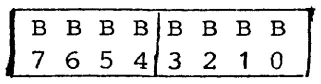




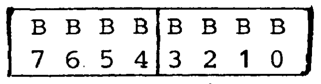
Operand or Address

Three_Byte Instruction





High Address or Skipped Byte 1



Low Address or Skipped Byte 2

APPENDIX C

An ICS Instrumentation Language

A command language which can be used for interactive instrumentation of an ICS has been developed. The purpose of this language is to enable the user to put instrumentation test points and set simulator conditions into the interpretive simulation of his machine code program giving him control over the simulation execution and the output. The language facilitates quick and easy debugging of machine code segments.

Before discussing command formats, a few terms must be defined. LOCATION will be used to denote a single location (LOC) and LOCATION* will be used to denote either a single location (LOC) or a range of locations (LOC1..LOC2). A location can be specified by a symbol (SYMB), an integer (100), or an expression (SYMB+3), i.e., offset from the symbol by three words. Symbols are included by inputting an alphabetically ordered symbol table. Integers may be in binary, octal, decimal, or hexadecimal. VALUE will be used to denote the values to be assigned to the specified location or locations. Values may be expressed as a single integer (INT) with a predefined multiplier of 1, a pair of integers with the multiplier denoting the number of values (MUL*INT), or a series of values (INT;MUL*INT;INT).

Examples of LOCATION*=VALUE:

B'101'=2	means	LOC 5 = 2
(2+3)*5=0	means	LOC 25 = 0
O'12'..O'14'=3*1	means	LOC 10,11,12 = 1
3..8=2*0;1;0;2*1	means	LOC 3,4,6 = 0 and LOC 5,7,8 = 1

The commands are divided up into four groups according to command types. Brackets, , denote that the contents where applicable can be repeated a number of times. Multiple commands may be placed on a line. A command and its parameters may not exceed column 72. Commands may be continued on the next line, however. A discussion of the commands and their formats follows.

I. Parameterless and Simple Commands

1. (Quit,Save)

Quit;

Quit - Causes the simulated execution to terminate.

Save - Causes the contents of all the registers and memory to be saved enabling a restart of the system at a later date.

2. (Start)

```
Start(LOCATION);

Start;
```

Start - Causes the execution to begin at the specified location.

Start(325);	Set the program counter to 325 and begin simulation.
Start;	Resume simulation using current program counter value.

II. Instrumentation Setting Commands

1. (Brtrace,-Brtrace,Checkpt,-Checkpt,Dump,-Dump, Halt,-Halt,Protect,-Protect,-Testval,Trace,-Trace)

```
Brtrace(LOCATION* ,LOCATION* );
```

Brtrace,-Brtrace - Sets the trace bits of all jump instructions to "on" and "off" respectively within the location range specified. Tracing the program execution path and the results of the executed statements can be formulated using this command.

Checkpt,-Checkpt - Sets the checkpoint bits to "on" and "off" respectively at the specified locations enabling users to determine register values at that point in the program.

Dump,-Dump - Sets and resets the dump bit at the specified locations enabling the user to selectively display memory contents each time those locations are reached.

Halt,-Halt - Sets and resets the halt bit at the specified locations. Halt causes the simulation to stop when a location whose halt bit is set has been reached. It is intended to keep a program from executing in an undesirable area. Upon Halt, control is returned to the user who may issue more commands for further execution, end, or resume the simulation. -Halt reverses this effect.

Protect,-Protect - Sets the protect bits to "on" and "off" respectively at the specified locations enabling the user to determine when and how critical data areas in his program are being used.

-Testval - Sets the test bits to "off" at specified locations and delinks those locations and their bounds from the test list. -Testval enables the user to cease testing whether or not the value at a particular location lies between the bounds specified by a previously executed Testval command.

Trace,-Trace - Sets and resets the trace bits respectively at the specified locations allowing tracing of the program execution path.

Checkpt(O'12'+2,20..27);	Set checkpoint bits at locations 12 and 20-27 to "on".
-Checkpt(22..24);	Set checkpoint bits at locations 22-24 to "off".

2. (Testval)

```
Testval(LOCATION,LB..UB ,LOCATION,LB..UB );
```

Testval - Sets the test bits at the specified locations to "on" and adds the location and its bounds to the test list. When locations whose test bits are set to "on" are reached by the simulation, a search of the linked test list is performed. Once the proper location is found, a test is made to determine if the value at that location is within the limits specified by the test list.

Testval(51,0..2,73,1..9);	Set test bits at locations 51 and 73 to "on" and the limits of those locations, 0-2 at loc. 51 and 1-9 at loc. 73, to the test list.

III. Memory and Register Commands

1. (Set,Fltreg,Fxreg,Statreg)

```
Set(LOCATION*=VALUE ,LOCATION*=VALUE );
```

<u>Set</u> - Sets memory locations to the values given in the command.

<u>Fltreg</u> - Sets the appropriate floating point registers to the values specified in the command.

<u>Fxreg</u> - Sets the appropriate fixed point registers to the values specified in the command.

<u>Statreg</u> - Sets the appropriate status registers to the values given in the command.

Set(5=2,10..14=2*0;3*1);	Set location 5 to 2, locations 10 and 11 to 0, and 12-14 to 1.
Fxreg(0..2=3*0;3=10);	Set fixed registers 0-2 to 0 and fixed register 3 to 10.

IV. Operating Commands

1. (Timeout)

```
Timeout(TIME);
```

<u>Timeout</u> - Causes the simulated execution to stop if the clock exceeds timeout time and returns control to the user. TIme is a real number and is normally interpreted in microseconds.

Timeout(100.0);	Stop when clock time exceeds 100.0 ms.

2. (Tdump)

```
Tdump(TIME,LOCATION* ,TIME,LOCATION* );
```

Tdump - The contents of the specified locations are dumped onto DUMPFILE when the clock time equals or exceeds TIME.

Tdump(195.5,A+3..450);	At a clock time of 195.5 ms, a dump of the contents of memory represented by 3 + symbol A's location through location 450 inclusive is made.

Two other operating commands, Ioreq and Interpt, may be entered into the system via OPFILE, the operating command text file. Tdump and Timeout commands may also be entered into the system through OPFILE as an alternate to the previously mentioned interactive method. OPFILE commands may extend over a line and multiple commands may appear on a single line. The layout of each of these commands on OPFILE follows.

```
Timeout TIME

Tdump STARTLOC COUNT TIME

Interpt CHANNEL COUNT TIME1 TIME2

Ioreq CHANNEL COUNT TIME1 TIME2 MUL VAL MUL VAL
```

TIME1 and TIME2 are real numbers indicating the time interval during which an operating file command is valid. CHANNEL is the channel number of which the I/O request must be honored. A description of Interpt and Ioreq follows.

Interpt - If the clock time is less than or equal to TIME2 and greater than or equal to TIME1, then an interrupt may occur based on the ability of the machine to handle an interrupt and the machine status. For example, in the NSSC-I, the result of an occurrence of an interupt depends on whether or not the interrupt override is set and if an interrupt of higher priority has not occurred at the same time.

Ioreq - An I/O request may be satisfied, based on the machine interrupt capability and the machine status, if the clock time is within the limits, TIME1 and TIME2. In the case of the NSSC-I, if the channel over which the cycle steal or program controlled I/O is active and no other I/O requests of higher priority are pending, then an I/O request can be satisfied. MUL and VAL are used to denote the values to be input into memory. COUNT denotes the number of values to be input or output. Inputs into or outputs from memory occur at the data block associated with each channel.

The syntax analyser has a built-in error recovery system enabling the user to correct mistakes made when inputting the interactive commands. The error is displayed on the succeeding line by " (error number)" indicating the location of the error and its type. An error within a command will inhibit it from inclusion into the test conditions. When an error occurs, the user should repeat the command. The error numbers and the corresponding error messages are listed below.

1: Instruction argument or (missing.

2: Too many arguments.

3:) missing.

4: Value expression is missing.

5: Argument not valid or ; missing.

6: Error in location interval.

7: Improper separator.

8: Operation symbol missing.

9: ' missing.

10: Improper number designator.

11: Improper numeric symbol.

12: Illegal number system.

13: Improper value expression.

14: Location range error.

15: Locations and values not equal.

16: Incorrect instruction.

17: Unknown symbol.

A sample command language input with deliberate errors has been provided to demonstrate the error handling. Syntax diagrams have also been provided to aid in the understanding of this simple, easy-to-use command language. This command language has proven to be extremely beneficial as an aid in debugging the I3CS system itself as well as the assembly language programs simulated by I3CS.

Sample Command Language Input With Deliberate Errors

```
? Dump;  Start(10..20);  Haltt(5);  Checkpt(10.20);
      ^ 1            ^ 2      ^16                ^ 6
? Halt(A'101');  Halt(0'101');  Trace(O'1A');  Halt(B'1011);
        ^12 ^ 8^ 3         ^12 ^ 8^  3     ^11             ^
9
? Set(5..10=3*2;2*1);  Set(2..3=2*5;2);  Set(5..8=3*1,2);
                   ^15               ^               ^15^ 4
? Halt(B'101G';ABC);  Halt(B'1011'+ABC);  Hlt(B'101';3);
             ^11^ 7^ 3 ^16                ^17    ^16     ^16
? Halt(5;3);
        ^ 7^ 3^16
? Trace(ABC);  Dump(100);  Protect(5..10,15;  Set(2*(1+2,9);
           ^17         ^12^ 7              ^ 7^ 3       ^ 3^
4^ 4
? Set(10);
        ^ 4
? Fxreg(1=O'400000');  Start;
```

APPENDIX D

NSSC-I Sample Run

```
? Trace(O'1514'..O'1515');
? Checkpt(O'471',O'1514'..O'1515');
? Statreg(1=O'7000'); Start(O'367');

     SIMULATION START TIME = 28335 MS
     SIMULATION END TIME = 28454 MS

  TRACE TIME=1532.00 IC= 845 ACCUM=030000 EXT=000000 INDX=000000
 EFFECT ADDR=000043 EFFECT ADDR VAL=001524
MNEMONIC=BRC

CHECKPT TIME=1532.00 IC= 845 ACCUM=030000 EXT=000000 INDX=000000
 STATUS REG=000000 STORAGE REG=007000 INTERRUPT OVERRIDE=000000
MNEMONIC=BRC

  TRACE TIME=1528.00 IC= 844 ACCUM=030000 EXT=000000 INDX=000000
 EFFECT ADDR=000046 EFFECT ADDR VAL=030000
MNEMONIC=TAE

CHECKPT TIME=1528.00 IC= 844 ACCUM=030000 EXT=000000 INDX=000000
 STATUS REG=001000 STORAGE REG=007000 INTERRUPT OVERRIDE=000000
MNEMONIC=TAE

CHECKPT TIME=159.00 IC= 313 ACCUM=200000 EXT=000000 INDX=000000
 STATUS REG=000000 STORAGE REG=007000 INTERRUPT OVERRIDE=000000
MNEMONIC=LDA


? Quit;


     SIMULATION START TIME = 28692 MS
     SIMULATION END TIME = 28693 MS
```

NOTES

1 B. Chatterton, Software Simulation of the Minuteman D17B Computer, NTIS No. AD-742-965, (Government Report Annul, Vol. 73, No. 14, 1973).

2 Anon., Microprocessor Support Software, Intel 4004, 4040, 8008, 8080 Simulators, Publication No. 76077900, (Minneapolis, Minnesota: Control Data Corporation, 1976).

3 Anon., Program Design Language Reference Guide (Processor Version 3), 4th edition, (Pasadena, California: Caine, Farber and Gordon, Inc., 1977).

4 Stephen H. Caine and E. Kent Gordon, "PDL - A tool for software design," (National Computer Conference, 1975), p. 271.

5 Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design, (Reading, Massachussets: Addison-Wesley Publishing Co., 1977), p. 82.

6 Ibid., p. 180.

BIBLIOGRAPHY

Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Reading, Massachussets: Addison-Wesley Publishing Co., 1977.

Anon. Microprocessor Support Software, Intel 4004, 4040, 8008, 8080 Simulators. Publication No. 76077900. Minneapolis, Minnesota: Control Data Corporation, 1976.

Anon. Program Design Language Reference Guide (Processor Version 3). 4th edition. Pasadena, California: Caine, Farber and Gordon, Inc., 1977.

Anon. RCA 1800 Microprocessors, User Manual for the CDP 1802 COSMAC Processor. Sommerville, New Jersey: RCA Solid State, 1977.

Caine,Stephen H. and E. Kent Gordon. "PDL - A tool for software design," National Computer Conference, 1975.

Chatterton, B. Software Simulation of the Minuteman D17B Computer. NTIS No. AD-742-965, Government Report Annul, Vol. 73, No. 14, 1973.

Jensen, Kathleen and Niklaus Wirth. Pascal User Manual and Report. New York: Springer Verlag, 1974.

VITA

Sharon Otero Beskenis

Born in Cheverly, Maryland, May 16, 1952. Graduated from Kecoughtan High School in Hampton, Virginia, June 1970. Attended Mary Washington College of the University of Virginia, 1970 - 1971 with a concentration in Pre-Foreign Service studies. Received a B.S. in Management Information Science, Christopher Newport College, 1973. Employed by Kentron International, Inc. as a senior software analyst. M.S. candidate, College of William and Mary, 1976 - 1980, with a concentration in Applied Science.