SIMULATION OF THE IBM SYSTEM/7

THESIS

Presented to the Graduate Council of the North Texas State University in Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Ted C. Lewis, Jr., B. S.
Denton, Texas
May, 1977
Lewis, Ted C., Jr., *The Simulation of an IBM SYSTEM/7*, Master of Science (Computer Sciences), May, 1977, 135 pages, 2 tables, 33 illustrations, bibliography, 18 titles.

This thesis describes the simulation of the IBM SYSTEM/7. The research leading to this thesis involved the development of a PL/I computer program that runs on an IBM 360/50 computer and simulates the IBM SYSTEM/7.

Various methods of simulation are examined and guidelines for computer simulation of another computer are established. The SYSTEM/7 simulator (SIM/7) is the heart of this thesis. SIM/7 simulates the IBM SYSTEM/7 entirely with software as opposed to an emulator which involves the combined use of hardware and software to perform the simulation process.

This thesis contains a general introduction to computer simulation, reason for simulation, a user's guide for SIM/7 and a definition of the SYSTEM/7 processor using the Vienna Definition Language.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LIST OF TABLES</td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td>LIST OF ILLUSTRATIONS</td>
<td>v</td>
</tr>
<tr>
<td>I</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Simulation Defined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simulation Techniques</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interpretation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Translation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Emulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>History of Simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reasons for Simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Service Functions</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>DESIGN CONSIDERATIONS</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Goals of the Project</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The Target Machine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The Implementation Language</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU Design Considerations</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>SYSTEM/7 PROCESSOR MODULE</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Processor States</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSTEM/7 Architecture</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction Classes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program Indicators</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSTEM/7 Instruction Set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Single-Word Operand Addressing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instructions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-Addressing Instructions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Instructions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Double-Word Operand Addressing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instructions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conditional Skip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conditional Branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State Control Instructions</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV. LOADERS</td>
<td>53</td>
</tr>
<tr>
<td>Compile-and-Go Loaders</td>
<td></td>
</tr>
<tr>
<td>General Loader Scheme</td>
<td></td>
</tr>
<tr>
<td>Absolute Loaders</td>
<td></td>
</tr>
<tr>
<td>Relocating Loaders</td>
<td></td>
</tr>
<tr>
<td>Direct Linking Loaders</td>
<td></td>
</tr>
<tr>
<td>SYSTEM/7 Loader</td>
<td></td>
</tr>
<tr>
<td>V. THE DEFINITION OF SIM/7 USING THE VIENNA DEFINITION LANGUAGE</td>
<td>66</td>
</tr>
<tr>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>The Vienna Definition Language</td>
<td></td>
</tr>
<tr>
<td>The Data Set</td>
<td></td>
</tr>
<tr>
<td>Operations Over the Data Set</td>
<td></td>
</tr>
<tr>
<td>The Mutation Operator</td>
<td></td>
</tr>
<tr>
<td>Relationships Between Objects</td>
<td></td>
</tr>
<tr>
<td>Instruction Execution</td>
<td></td>
</tr>
<tr>
<td>A VDL Description of the IBM SYSTEM/7 Simulator</td>
<td></td>
</tr>
<tr>
<td>Representations of Registers</td>
<td></td>
</tr>
<tr>
<td>Register Functions</td>
<td></td>
</tr>
<tr>
<td>Effective-Address-Generation State</td>
<td></td>
</tr>
<tr>
<td>The Execute State</td>
<td></td>
</tr>
<tr>
<td>VI. HOW TO USE THE SYSTEM/7 SIMULATOR</td>
<td>95</td>
</tr>
<tr>
<td>Support Software</td>
<td></td>
</tr>
<tr>
<td>Trace Control Card</td>
<td></td>
</tr>
<tr>
<td>Trace of Execution</td>
<td></td>
</tr>
<tr>
<td>Full Trace</td>
<td></td>
</tr>
<tr>
<td>Partial Trace</td>
<td></td>
</tr>
<tr>
<td>Loader Input Trace</td>
<td></td>
</tr>
<tr>
<td>Branch and Link Trace</td>
<td></td>
</tr>
<tr>
<td>Error Messages</td>
<td></td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

(Continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VII. MAINTAINING THE SYSTEM/7 SIMULATOR</td>
<td>110</td>
</tr>
<tr>
<td>Modifying the Program</td>
<td></td>
</tr>
<tr>
<td>Program Description</td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td></td>
</tr>
<tr>
<td>Functional Description of Instructions</td>
<td></td>
</tr>
<tr>
<td>VIII. CONCLUSIONS</td>
<td>116</td>
</tr>
<tr>
<td>Evaluation of the Simulator</td>
<td></td>
</tr>
<tr>
<td>Areas for Future Development</td>
<td></td>
</tr>
<tr>
<td>APPENDIX</td>
<td>122</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>125</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. SYSTEM/7 Opcodes and Execution Times</td>
<td>36</td>
</tr>
<tr>
<td>II. Instruction Characteristics</td>
<td>115</td>
</tr>
</tbody>
</table>
### LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A SYSTEM/7 Word</td>
</tr>
<tr>
<td>2.</td>
<td>Range of Numeric Values</td>
</tr>
<tr>
<td>3.</td>
<td>Processor Data Flow</td>
</tr>
<tr>
<td>4.</td>
<td>Short-Format Instruction Word</td>
</tr>
<tr>
<td>5.</td>
<td>Non-Addressing Instruction Word</td>
</tr>
<tr>
<td>6.</td>
<td>Shift Instruction Word</td>
</tr>
<tr>
<td>7.</td>
<td>Double-Word Operand Addressing Instruction Word</td>
</tr>
<tr>
<td>8.</td>
<td>Conditional Skip Instruction</td>
</tr>
<tr>
<td>9.</td>
<td>Condition Code Masks</td>
</tr>
<tr>
<td>10.</td>
<td>Extended Mnemonics</td>
</tr>
<tr>
<td>11.</td>
<td>Conditional Branch Instruction Format</td>
</tr>
<tr>
<td>12.</td>
<td>Compile-and-Go Loader Scheme</td>
</tr>
<tr>
<td>13.</td>
<td>General Loader Scheme</td>
</tr>
<tr>
<td>14.</td>
<td>Processing a SYSTEM/7 Program Which Does Not Require Linkage with Other Modules</td>
</tr>
<tr>
<td>15.</td>
<td>Processing SYSTEM/7 Programs Using the Linkage Editor</td>
</tr>
<tr>
<td>16.</td>
<td>Example of Object Deck for Link/7</td>
</tr>
<tr>
<td>17.</td>
<td>Flow Diagram of Loader for SYSTEM/7 Simulator</td>
</tr>
<tr>
<td>18.</td>
<td>A Simple Object</td>
</tr>
<tr>
<td>19.</td>
<td>A Composite Object</td>
</tr>
<tr>
<td>20.</td>
<td>The Mutation of an Object</td>
</tr>
</tbody>
</table>
**LIST OF ILLUSTRATIONS**  
*(Continued)*

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.</td>
<td>The Structure of Control Trees</td>
<td>73</td>
</tr>
<tr>
<td>22.</td>
<td>Processor Data Flow</td>
<td>77</td>
</tr>
<tr>
<td>23.</td>
<td>Job Control Language for Assemble, Link, Format, and Execute</td>
<td>97</td>
</tr>
<tr>
<td>24.</td>
<td>Trace Parameter Card</td>
<td>98</td>
</tr>
<tr>
<td>25.</td>
<td>Trace Control Card Example</td>
<td>99</td>
</tr>
<tr>
<td>26.</td>
<td>Full Trace of Sample Program in the Appendix</td>
<td>101</td>
</tr>
<tr>
<td>27.</td>
<td>Partial Trace of Sample Program in the Appendix</td>
<td>103</td>
</tr>
<tr>
<td>28.</td>
<td>Loader Dump for Sample Program in the Appendix</td>
<td>104</td>
</tr>
<tr>
<td>29.</td>
<td>Branch and Link Table for Program in the Appendix</td>
<td>105</td>
</tr>
<tr>
<td>30.</td>
<td>Input Macro</td>
<td>106</td>
</tr>
<tr>
<td>31.</td>
<td>Output Macro</td>
<td>107</td>
</tr>
<tr>
<td>32.</td>
<td>Comparison of SYSTEM/7 Errors and SIM/7 Errors</td>
<td>108</td>
</tr>
<tr>
<td>33.</td>
<td>Flow Diagram of SIM/7</td>
<td>113</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

This thesis is a proposed solution to the problem of simulating an IBM SYSTEM/7. The research leading to this thesis involved the development of a PL/I computer program that runs on an IBM 360/50 computer and simulates an IBM SYSTEM/7. A detailed examination of the SYSTEM/7 simulator (SIM/7) will be presented, but first a few introductory remarks regarding simulation in general are appropriate.

Simulation Defined

Computer modeling and simulation is a powerful technique with a broad range of applications in research and analysis activities. The techniques of computer simulation have had a remarkable development, and speculation about its future applications are equally impressive. However, the definition of computer simulation seems to depend on whoever is discussing the technique, and what their particular application is.

The layman may understand very little about computers or computer simulation, and may view the entire process with skepticism. A statistician may view the technique as a synthetic experiment with flexibilities not possible in
laboratory or field experimentation. The mathematician may view computer simulation as a tool of mathematical logic and computation. Systems analysts may see computer simulation as only one of several simulation techniques. A programmer may view computer simulation as a challenging programming problem, with no interest in what the end results are.

Actually, the definition of computer simulation includes all of these viewpoints. It is a synthetic experiment, a mathematical tool, a simulation technique, and a programming problem; however, for particular applications a more specific definition is usually appropriate.

Caplener (1) defines simulation as the process of conducting experiments on a model of a system in lieu of either direct experimentation with the system itself, or direct analytical solution of some problem associated with the system. Webster (6) defines simulation as "the representation of a system by a device (such as a computer) that imitates the behavior of the system." An exhaustive list of the different definitions of and types of simulation would be virtually endless. Since simulation techniques are applied so differently and in so many diverse fields, the word "simulation" has different meanings for persons of different backgrounds. This thesis is concerned with one particular type of simulation (simulation of one
computer by another computer); however, it is perhaps appropriate to discuss briefly a few other kinds of simulation.

A discrete event system is defined as any system that can be described by a sequence of discrete events. Examples of discrete event systems include digital computers, production lines, and queues. An event is something that causes a change in the state of the system. For example, the Europeans form queues at taxi stops. The arrival of a new person would constitute an event. An event occurs at an instant in time and immediately changes the state of the system. Events may occur at random (e.g., the arrival of a person at a queue) or at fixed intervals (e.g., the command to fetch an instruction in a computer system).

Some systems change continuously rather than at discrete intervals. A simulator for such a system is called a continuous system simulator. Continuous systems may be described by mathematical equations. Continuous system simulators using digital computers employ a digital approximation of these equations. One example of a continuous system is the flow of jobs through a computer center. The state of the computer center changes continuously as jobs flow from one work station to another. All jobs do not arrive at a station at one instant but gradually move
through the computer center. The functioning of the human body (blood flow, growth, etc.) could be modeled as a continuous system.

Simulation Techniques

There are at least three generally accepted techniques for computer simulation. They are:

1. INTERPRETATION
2. TRANSLATION
3. EMULATION

Each of the techniques is examined below.

Interpretation

Sammet (5) defines an interpreter as a program which executes a source program, usually on a step-by-step, line-by-line, or unit-by-unit basis. In other words, the interpreter executes the smallest possible meaningful unit in the program. A simulator-interpreter therefore uses a software routine to perform the action of each instruction it decodes. Since an instruction is decoded each time it is executed, interpreters are probably the slowest technique used in simulation; however, interpreters are attractive for several reasons. An interpreter can execute self-modifying programs. Also, interpretation is a single step process, while translators require two steps to simulate a program.
Interpretation lends itself to an educational environment, where much time is spent in the debugging process. An interpreter has control over the execution of the simulated program and can perform powerful debugging functions that are not always available with the hardware. For example, an interpreter lends itself to a step-by-step account (trace) of the simulated program execution. An interpreter can also be programmed to check for some specified condition before executing each instruction. This allows the use of utility routines, such as a memory or register dumps, which are valuable debugging tools. An interpreter can also be programmed to detect such errors as memory references outside the boundary limitations and other errors which may not be detected by the hardware computer.

This thesis uses an interpreter to implement the SIM/7 simulator. So, the following chapters focus primarily on interpretation as a technique of computer simulation.

Translation

A translator is a program which translates a program written for the target machine into the language of the host machine. The translated program can then be executed on the host machine. Two distinct steps are involved in the translation. The first step inputs the code written
for the target machine and generates equivalent code for the host machine. The function of each instruction in the target machine's language is duplicated by an instruction or a series of instructions in the host computer's instruction set. This is similar to the way a macro-assembler handles macro calls. The second step is the execution of the translated program by the host computer.

One advantage of translation is that the translated code is in the language of the host computer and can be executed efficiently. Also, the second step can be executed independently of the first step, as long as the program is not changed.

A disadvantage of translators is that self-modifying programs cannot execute properly. Also, translators do not lend themselves to the use of utilities or step-by-step traces of instruction execution. The problem is that the execution is performed by the host computer hardware and is not software controlled.

Emulators

A simulator that is implemented by micro-programming the control unit of a computer is called an emulator. The micro-instructions are stored in a read-only memory (ROM). The microprogram located in the ROM controls the operation of the computer, so the characteristics of the computer
can be changed by altering the microprograms located in the ROM. To emulate a computer system, the control unit of the host is microprogrammed to execute the instructions of the target computer. The host computer then assumes the identity of the target computer.

Emulation is an extremely fast technique for simulation. It is probably the best technique for production type systems; however, there are some disadvantages to emulation. Since emulation does not lend itself to user utilities and the tracing of instruction execution, it is not suitable for an educational environment. Another disadvantage is that only a microprogrammable computer can be used as the host for an emulator. Also, when an emulator is to be run, the control unit of the host must be changed. This does not allow execution of normal programs until the control unit is changed back to its original form. Since the ROM must be changed, it is difficult to modify an emulator. Therefore, if the simulator is to be very dynamic, emulation is not practical.

History of Simulation

The verb "to simulate" is a term that has become popular recently in a number of disciplines to describe the ancient art of model building. Computer, or at least electronic circuit, simulation began as early as 1943 with
a study of the dynamics of aircraft and guided bombs (2). After World War II, analog computers (or electronic differential analyzers or simulators) began to emerge from the shadows of security classification. The modern use of the word, however, traces its origin to the work of Von Neumann and Ulam in the late 1940's, when they coined the term "Monte Carlo Analysis" to apply to a mathematical technique they used to solve certain nuclear-shielding problems that were either too expensive for experimental solution, or too complicated for analytical treatment.

"Monte Carlo Analysis involved the solution of a non-probabilistic mathematical problem by simulating a stochastic process which has moments or probability distributions satisfying the mathematical relations of the non-probabilistic problem." (3).

With the introduction of the high-speed computer in the early 1950's, simulation took on still another meaning because it had become possible to experiment with mathematical models on a computer. Controlled, laboratory-like experiments could now be performed; however, they used the electronic computers rather than physical devices such as a nuclear reactor.

One of the early applications of computer simulation was in the modeling of computer logic. Simulation programs assisted in the hardware design process, verifying the
accuracy, completeness, or performance of logical designs, and provided a cheap and easy means to test the effectiveness of new design ideas. Ragazzini (4) stated that "step-by-step computer simulation of digital hardware as it is designed allows its internal performance as well as its interaction with other systems to be predicted before any hardware is built."

As computer hardware became more complex, simulation was found to be an excellent means for obtaining more general information about the behavior of the system as a whole. These types of needs have persisted, and simulations are still being employed effectively in this regard. On the whole, models developed for this purpose have been very successful and have demonstrated their worth. As hardware has developed from one generation to the next, this type of simulation approach has continued to be employed in the basic or logical design of circuitry, and it is expected that the usage will persist in the future.

In the early days of computer usage, the primary focus of attention was upon the hardware, if only because there was very little software. As experience was gained in the use and application of computer systems, a number of software packages emerged. First came assemblers, then special purpose routines, rudimentary compilers, more comprehensive compilers, etc. Before long, the need for operating systems arose.
The more complicated specialized operating systems that were developed gave rise to a host of simulations. Many of these were constructed after the software system in question, and were used for the purpose of evaluating proposed modifications to that system. Others were developed before the system had been designed or programmed and were used to aid in the development of the system as well as its subsequent modification. On the whole, these simulations were quite successful. They were able to reflect the various systems with reasonable accuracy and were able to satisfy their own design goals surprisingly well.

More recently, the use of computer-simulation-of-computers has been used, in industry, as a means of programming a system before actually having the hardware available, and in education as a teaching tool when desired hardware is not available.

An example of a simulator being used in an educational institution is the HMS 5050 Computer System Simulator designed by the computer science students at Michigan State University (7). The HMS 5050 Computer System was designed to give students access to a computer which had an interrupt system and programmable data channels. The HMS 5050
Simulator enables students to implement a major component of an operating system. Without the use of the simulator, this would not be possible.

Reasons for Simulation

This section examines some of the advantages of computer simulation. Specific questions about a system which may be in its conceptual phase, in a phase of development and testing, or already built often can only be answered by computer simulation. Assuming that the model is valid, in that it accurately simulates the system, the model is an inexpensive, efficient, rapid device to analyze and evaluate a system.

Following is a list of some of the reasons for using computer simulation.

1. Simulation makes it possible to study and experiment with the complex internal interaction of a given system, even though access to the actual system is not possible. For example, economists do not have the capability of experimenting with the economy, but they can do so using models.

2. Through simulation, one can study the effects of changes on the system by making alterations in the model of the system and observing the effects of the changes. This is done in the space industry where they can simulate
conditions of outer space and see how changes in the various conditions effect their spacecraft.

3. Detailed observations of a system being simulated may lead to a better understanding of the system. This could lead to improvements which otherwise may not be obtainable.

4. Simulation can be used as a device for teaching both students or practitioners basic skills in programming, systems analysis, or hardware design.

5. Simulation can serve as a method of preparing for a new system. Software can be developed before the system is delivered.

6. Simulation is a convenient way to break down a complicated system into subsystems, each of which can be analyzed individually.

7. When new components are introduced into a system, simulation can aid in the forecast of potential problem areas.

8. Simulation can provide better error diagnostics than the actual system (See Chapter VI).


10. In most cases, it is easier to obtain access to a simulator than to the actual hardware.

12. Simulation gives the analyst better insight into the system as a whole.

The primary reason for the development of the SYSTEM/7 Simulator (SIM/7) was for use in the testing of application programs in the absence of a test system. The facilities incorporated in SIM/7 (the detailed tracing of instruction execution, extended error analysis, precise timing, and trace-back capability) also make it a valuable educational tool. Its modular design also affords the opportunity for modifications to the system. This could prove to be a valuable tool in the study of minicomputer design and architecture.

Service Functions

Several service functions are provided by SIM/7. (A full discussion of each function is included in Chapter VI.) The service functions are as follows:

1. The Trace Function allows execution to be traced at the following levels:
   
a. Full Trace - this includes a detailed output of everything happening in the system. The full trace can be obtained over either a span of time or a region in memory.
b. Partial Trace - this includes only the location of the instruction being executed, the decoded instruction, and the time. The partial trace can be obtained over either a span of time or a region in memory, just as the full trace.

2. The Loader Dump allows the dumping of the binary input to the loader. This function is useful only in debugging a new modification to SIM/7.

3. The Snap Dump function is software callable and allows the dumping of specified areas of memory.

4. Input/Output routines are software callable and allow the input/output of data.

5. The Loader automatically loads the program into memory at the beginning of execution.

6. The Branch and Link Table is automatically output at the end of each run and provides a trace-back facility.

The service functions are extremely valuable to the student or beginning programmer; they give much needed control over the program execution and are valuable debugging tools.
CHAPTER BIBLIOGRAPHY


CHAPTER II

DESIGN CONSIDERATIONS

There is a great deal more to writing a simulator than just coding an interpreter. Before any code is written, the basic design considerations must be considered. Chapter II examines the various problems as they apply to the SYSTEM/7 Simulator, and seeks to find the best possible solution.

Goals of the Project

The first step in implementing any type of system is to establish the goals that are to be accomplished. This is of particular importance in writing a simulator, since it is to effectively simulate the workings of a target machine. Different simulators are used by different users and in different ways, so not all simulators should be designed in the same manner. The type of application and the level of knowledge of the user must be primary considerations. For example, a simulator which is used for production type work ought to be coded primarily for speed. Since production programs are assumed to be working programs, extra error diagnostics or informative messages are not necessary. In fact, they would be counter-productive to a production system; however, if the simulator is to be
written for the beginning or student programmer, speed becomes a secondary objective. Extensive error-detection capabilities with self-explanatory messages and extensive debugging facilities become a necessity if the simulator is to be used as a training tool. The simulator developed for this thesis is for use as a training tool for beginning or student programmers, and as a means of developing and testing routines useful in industry; however, it is not for use in executing programs in a production environment.

The design considerations of the simulator dictate the extent to which the target machine is simulated. Since the SYSTEM/7 simulator is primarily a training tool for beginning programmers, no I/O instructions are simulated. The I/O instructions for the SYSTEM/7 are extremely difficult to code at the assembler level. Therefore, it would be of little practical use to implement these instructions; however, if the simulator were being written for use by highly trained analysts for test or production runs, it would be a necessity to implement the I/O instructions.

In order to measure the effectiveness of the implemented simulator, it must be weighed against the original design objectives. The SYSTEM/7 simulator was designed with the following goals in mind.
1. Good support software is a vital requirement for a successful simulator. The support software should include an assembler, linkage editor (loader), utility routines (service functions), and extensive debugging routines. The support software should be kept as simple and easy to use as possible.

2. A simulator is of practical use only as long as it is kept current, i.e., up-to-date with latest system developments. Therefore, an easy method of modifying and updating the support software is a vital necessity.

3. The user interface should be as simple as possible and as close to the actual system as possible. If the user has to spend the majority of his time trying to gain access to the system, the time available for learning to program the system is diminished.

4. The simulator should be coded to achieve the maximum speed for the particular application.

5. The simulator should be coded to occupy a minimum amount of memory, without sacrificing speed. In general, the smaller the core size of a program, the faster it will be brought to a control point for execution. This is especially true if the program is run concurrently with larger production programs.

6. The simulator should execute every possible sequence of instructions. Through intelligence and/or
ignorance, beginning programmers will at times generate sequences of code which were not expected. Therefore, it is of the utmost importance that every possible sequence of instructions execute properly in order to maintain the integrity of the simulator.

7. The simulator should be foolproof. That is, there should be no possible way to "hang up" the system or "crash" the simulator. The simulator should not depend upon the programmer to do anything properly, but should detect any and all errors.

8. The error messages provided by the simulator should be extensive and self-explanatory. The user should not have to dig through a book of codes to find an obscure message which does not lend itself to interpretation. (For example, refer to IBM's Messages and Codes for the IBM 360/50.)

9. The simulator should be as easy to modify as is practical. This should be a major consideration in choosing an implementation language.

10. Finally, the entire simulator system must be thoroughly documented. This includes both user documentation and program comments. Documentation is one of the most important parts of the simulator. Without documentation, the simulator is of no practical use.
The Target Machine

Before writing a simulator, the programmer must have a thorough knowledge of the target machine. He should have a thorough understanding of the instruction set and the hardware. Simply a working knowledge of a computer is not good enough. Nearly all computers have a few subtle features that the average programmer never discovers, or that he works around. For machines with large instruction sets, most programmers simply learn a subset of the language and never use many of the instructions.

The programmer writing a simulator must know enough about the target machine to handle any situation that may occur. For example, some machines will have two different instructions that generate the same opcode or have opcodes that are undefined. For some computers, undefined opcodes may default to some legitimate instruction, and for other computers, they may have unique effects. At any rate, the programmer must have the knowledge to understand how to handle any possible situation.

Normally, programmers' manuals do not contain enough information to adequately define a machine, so the programmer must seek information from other sources. Logic diagrams of the target machine would be helpful; however, they are not always easily understood by non-engineers.
Also, logic diagrams are not available for some machines, as was the case with the SYSTEM/7; however, if the target machine is available, the programmer can solve some problems by experimentation. This is not the best procedure for defining the characteristics of a particular machine, but it is better than not having any information at all. One manual which is usually a very good source of information in defining a simulator is the functional characteristics manual. If available, the functional characteristics manual for a target machine defines the instructions and how each instruction effects the registers and result indicators. This information is extremely helpful in defining the simulator.

Besides having a knowledge of the target computer's hardware and being experienced at programming the target computer, the programmer must be proficient at programming the host computer. This is necessary if the simulator is to be as fast as possible and still occupy as little memory as possible. In many cases, no one person will possess the knowledge of both the hardware and software required in writing a simulator. In this situation, a team of programmers will be used to design and write the simulator.

The Implementation Language

Choosing the right implementation language is a very important consideration in designing a simulator. The big
question is whether the simulator should be coded in
assembly language or in a high-level language. The type
of application and the type of user should be primary con-
siderations in determining what type of language to use.

If an extremely fast and efficient operating system
is the only goal, such as a production simulator, then the
simulator should definitely be written in assembly language
by an "expert" programmer. The term "expert" is used be-
cause simply coding a program in assembly language does not
make it fast and efficient. The code generated by modern
compilers is often more efficient than similar assembler
language code written by an "average" programmer/analyst.
An "expert" programmer can utilize the features of the host
more fully using assembly language. For example, if the
host machine has enough registers available, the programmer
can use the host's registers, instead of memory, to simulate
the registers of the target machine.

There are several disadvantages to using assembly lan-
guage, however. One problem is routine system changes
which may affect the assembly language program. Another
problem is that an assembly language program is machine-
dependent. An assembler program usually is not compatible
between different machines, even if the machines are built
by the same manufacturer. This greatly limits who can use
the assembler program. Assembler language programs are
also more difficult to modify, since debugging is more difficult than with high-level languages. The average student or beginning programmer is not capable of reading and understanding an assembler program, much less being able to modify it. Therefore, careful consideration must be given to the type of application and the type of user before using assembler language.

With the current state-of-the-art of compilers being as highly sophisticated as it is, an "adequately" efficient simulator can be written in a high-level language. It is an accepted fact that a more efficient simulator can be written in assembler language; however, the advantages of a high-level language far outweigh the speed gained by using assembly language. This is especially true if the simulator is to be used as a training tool for beginning programmers.

One of the primary advantages of using a high-level language is that the simulator program is compatible between machines in many cases. For example, the SYSTEM/7 simulator has been run successfully on the IBM 360/50 under OS and on the IBM 370/145 under VS with no modifications. High-level languages, in many instances, are compatible between machines built by different manufacturers. For example, IBM 360/50 FORTRAN programs are usually easily converted to run on the CDC 6600, a conversion from
a 32-bit word machine to a 60-bit word machine illustrates how a high-level language can make a simulator available to a wider range of users. In most cases, a high-level language is nearly immune to system changes and requires less modification to keep the system current. Debugging and modification are more easily accomplished using a high-level language, making it possible for beginning programmers to make temporary modifications to the simulator when studying the structure of the target machine. By doing so, they can learn the effects of software and hardware changes by experimentation. Ease of modification and debugging also makes the job of keeping the simulator current and adding new features a practical task. Last, but not least, a high-level language is self-documenting to some extent. This is an enormous advantage to someone trying to find out how the simulator handles a particular situation, or to someone trying to modify it.

There are many high-level languages available which would be suitable for preparing a simulation program, such as FORTRAN, COBOL, ALGOL, and PL/I. PL/I is by far the best language for use in simulation. The primary reason is the extensive bit-manipulation capabilities of PL/I. In PL/I, a fixed binary number is automatically 16-bits in length. This is an enormous advantage when simulating a target
machine with a word length of 16-bits. By declaring memory to be a fixed binary array, there is automatically an array of 16-bit words. This eliminates the problem of what to do with the excess bits in the host computer word.

From the above comments, it is probably apparent that the SYSTEM/7 simulator is written in a high-level language, PL/I in particular.

CPU Design Considerations

The control processor, or interpreter, is the heart of the simulator. It duplicates the functions of the target computer's Central Processor Unit (CPU). The interpreter presents some interesting coding problems and requires careful considerations.

The major problem that affects most simulators is that of word size. If the host machine has a smaller word size than the target machine, the simulator may be extremely difficult to code. Each memory word of the target machine must be represented by two or more words in the host computer. This means that a simple, one word operation on the target must be performed in multiple-precision on the host. This complicates such operations as arithmetic and addressing. For example, a simple subtraction would require a multiple-precision complement of a number. Multiple-precision routines are more difficult and lengthy
than single-precision routines. The complexity of such
operations makes the simulator slower and more difficult
to understand.

If the host computer's word is larger than the target
machine's word, then fewer problems exist. The main prob-
lem in this situation is what to do with the extra bits.
The simplest thing to do is ignore the extra bits; however,
if memory space is a problem, the extra bits may be used as
flag bits or as an overlay area for other tables, constants,
or variables. No matter how the extra bits are used, they
create problems. The extra bits must be masked either be-
fore or after nearly every operation. Other problems in-
volving the extra bits include how to handle the carry bit
from an arithmetic operation and how to handle sign exten-
sion. These and many other problems arise because the host
computer's word is longer than the target computer's word.

If the host computer's word size is a multiple of the
target computer's word size, then two or more words of
simulated memory may be packed into a single memory word
of the host. This packing scheme can save an enormous
amount of storage; however, unless storage is a problem,
this method is a questionable advantage. The complex pro-
cedure required for addressing significantly reduces the
speed of the simulator; however, if the host computer has
a byte-addressing feature, simulated memory can be accessed
through a single instruction. This combination of hardware-software could be used to efficiently pack simulated memory, since the hardware performs the addressing calculations. Problems may exist, however, in converting addresses from target-machine representation to byte-mode format.

The ideal situation is when the host computer's word is the same length as the target computer's word. In this situation very few, if any, problems exist; however, this is not normally the situation, since the target machine being simulated is usually simulated on a larger host machine which offers a larger word size.

PL/I features a 16-bit fixed binary (2) number. By declaring simulated memory to be a fixed binary array, the host machine appears to have the same word length as the target machine, (i.e., if the target machine features a 16-bit word). This greatly simplifies the coding and causes the modification and readability of the simulator to necessarily be simplified. This was one of the primary reasons for choosing PL/I as the implementation language.

Another CPU consideration is that of register assignment. By declaring the registers to be fixed binary (PL/I) words, they can be represented in memory as a 16-bit word. This is another illustration of how the 16-bit fixed binary word simplifies the coding of the simulator.
After resolving the problems of how to represent memory and registers, a series of other problems still exist. One common problem is when the target computer and the host computer use different methods for representing numbers. An example of this is the DLS-620 simulator (1) written by D. C. Pheanis. The most common types of numeric representations are two's complement, one's complement, and signed-magnitude. The simplest case to implement is when both machines are two's complement arithmetic. If either machine uses one's complement arithmetic, then the end-around carry must be dealt with. The end-around carry complicates arithmetic routines, such as the multiply and divide. If only one machine uses signed-magnitude arithmetic, then conversion routines will be required for arithmetic calculations.

The SYSTEM/7 simulator runs on a two's complement computer (IBM 360/50) and simulates a two's complement computer (IBM SYSTEM/7). This greatly simplifies the structure of the simulator and increases its efficiency.
CHAPTER BIBLIOGRAPHY


CHAPTER III

SYSTEM/7 PROCESSOR MODULE

This chapter describes the SYSTEM/7 processor module and instruction set in terms that are understandable to programmers. Since SIM/7 is intended for use by students or beginning programmers, it features a few error detection facilities that are not included in the real system. Those facilities are discussed whenever they cause a difference between the simulator and the CPU for the real machine. For example, a reference to a location outside the limits of memory will not be detected by the real CPU, but will be recognized as an error by the simulator. Since this chapter describes both the simulator and the real CPU in detail, it can be used as a programming manual for either.

Processor States

There are five possible states which may exist on the IBM SYSTEM/7 (Stop, Manual Wait, Wait, Execute, and Load States). Only four of these states, however, are pertinent to SIM/7. They are the Stop, Wait, Execute, and Load States.

The processor enters the Stop State and turns on the corresponding indicator (see description of full trace, Chapter VI) when the processor detects an error condition,
or when the stop (STP) instruction is executed. SIM/7 does not allow an exit from the state.

The Wait State is entered, and the corresponding indicator is turned on, when the level exit instruction (LEX) is executed. Since there is only one level of execution in SIM/7, there is no exit from this state.

The processor enters the Load State automatically when SIM/7 is executed. The programmer has no control over the Load State (the SIM/7 loader is discussed in Chapter IV). After completion of the Load State, the simulator enters the Execute State.

SYSTEM/7 Architecture

The IBM SYSTEM/7 has 4096 words of 16-bit core memory, and memory is expandable up to 32,768 words. Bits within a word are numbered from left to right (bits 0 through 15), as shown in Figure 1.

Figure 1 - A SYSTEM/7 Word
Bit zero (high-order bit) of a word is the sign bit. Memory words are addressed 0 through 7FFF (hexadecimal representation), as shown in Figure 2.

<table>
<thead>
<tr>
<th>Bit 0 (sign)</th>
<th>Bits 1-15 (numeric value)</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 0 to 111</td>
<td>000 000 0000 0000</td>
<td>00000</td>
<td>0000</td>
</tr>
<tr>
<td>- 1 to 111</td>
<td>000 000 0000 0000</td>
<td>-32,768</td>
<td>8000</td>
</tr>
</tbody>
</table>

**Figure 2 - Range of Numeric Values**

Positive numbers are represented in a true binary notation with a zero sign bit. Negative numbers are represented in two's complement binary form, which specifically excludes a negative zero.

The functions of the major registers of the SYSTEM/7 are to store, control or retrieve data in the system, to modify the data in memory, and to prepare instructions fetched from memory for interpretation. The Storage Data Register (SDR) acts as the intermediate storage register for interregister data transfers. Every word of data transferred to or from main storage by the processor passes through the SDR. The Accumulator (ACC) is used in most arithmetic and logical operations. The Storage Address
Register (SAR) is used by the processor to access each location in main storage. The Instruction Address Register (IAR) holds the address of the next instruction to be executed. The Operation Register (OR) holds the first 16 bits of the instruction fetched from storage. The Index Registers (XR(1) to XR(7)) are used in generating the Effective Address (EA) of an instruction or in arithmetic and logical operations. Each register is 16 bits in length. Since SIM/7 is written in a higher level language (PL/I), it uses memory locations to represent each of the registers. The diagram in Figure 3 shows the relationship of the registers and the rest of the computer's architecture. Chapter V gives a rigid definition of the SYSTEM/7 CPU using the Vienna Definition Language.

A basic problem in designing a computer with a 16-bit word is how to address up to 32,768 (Hex-7FFF) words of memory and still have enough bits left over in each instruction word to have a reasonably extensive instruction set. The problem is partly solved by using several different modes of addressing. Also, the instruction set is broken down into several classes; some classes are nonaddressing and therefore require no address field.

Addressing calculations are performed in 16-bit, two's complement arithmetic. There are several different
Figure 3 - Processor Data Flow
modes of addressing. The different modes will become apparent as the various instruction classes are discussed.

Instruction Classes

Functionally, the instruction set can be subdivided into six classes. The classes describe, in general, the type of data operation that is performed (1). The six classes are:

1. Load and Store
2. Arithmetic
3. Logical
4. Shift
5. Branch
6. Register to Register

Table I contains a complete list of mnemonic codes, operation codes, and execution times for each instruction; however, for purposes of simulation, the instruction set will be subdivided into seven classes. The classes describe the addressing techniques of the instructions. The seven classes are:

1. Single-word, operand addressing instructions
2. Nonaddressing instructions
3. Shift
4. Double-word, operand addressing instructions
5. Conditional skip
6. Conditional branch
7. State Control Instructions
## TABLE I

**SYSTEM/7 OPCODES AND EXECUTION TIMES**

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>MNEMONIC</th>
<th>OPERATION CODE</th>
<th>EXECUTION TIME IN NANOSECONDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD AND STORE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Accumulator</td>
<td>L</td>
<td>11000</td>
<td>800</td>
</tr>
<tr>
<td>Load and Zero</td>
<td>LZ</td>
<td>11001</td>
<td>1200</td>
</tr>
<tr>
<td>Load Immediate</td>
<td>LI</td>
<td>01100</td>
<td>400</td>
</tr>
<tr>
<td>Load Index Long</td>
<td>LXL</td>
<td>10001</td>
<td>1200</td>
</tr>
<tr>
<td>Store Accumulator</td>
<td>ST</td>
<td>11010</td>
<td>800</td>
</tr>
<tr>
<td>Store Index</td>
<td>STX</td>
<td>01101</td>
<td>800</td>
</tr>
<tr>
<td>ARITHMETIC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>A</td>
<td>10001</td>
<td>800</td>
</tr>
<tr>
<td>Subtract</td>
<td>S</td>
<td>10010</td>
<td>800</td>
</tr>
<tr>
<td>Add Register</td>
<td>AR</td>
<td>11111</td>
<td>400</td>
</tr>
<tr>
<td>Subtract Register</td>
<td>SR</td>
<td>11111</td>
<td>400</td>
</tr>
<tr>
<td>Complement Register</td>
<td>CR</td>
<td>11111</td>
<td>400</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>AI</td>
<td>01110</td>
<td>400</td>
</tr>
<tr>
<td>LOGICAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>N</td>
<td>11100</td>
<td>800</td>
</tr>
<tr>
<td>OR</td>
<td>O</td>
<td>11101</td>
<td>800</td>
</tr>
<tr>
<td>Exclusive OR</td>
<td>X</td>
<td>11110</td>
<td>800</td>
</tr>
<tr>
<td>AND Register</td>
<td>NR</td>
<td>11111</td>
<td>400</td>
</tr>
<tr>
<td>OR Register</td>
<td>OR</td>
<td>11111</td>
<td>400</td>
</tr>
<tr>
<td>Exclusive OR Register</td>
<td>XR</td>
<td>11111</td>
<td>400</td>
</tr>
<tr>
<td>SHIFTING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Left Logical</td>
<td>SLL</td>
<td>00010</td>
<td>400+50N</td>
</tr>
<tr>
<td>Shift Left Circular</td>
<td>SLC</td>
<td>00010</td>
<td>400+50N</td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td>SRL</td>
<td>00010</td>
<td>400+50N</td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>SRA</td>
<td>00010</td>
<td>400+50N</td>
</tr>
<tr>
<td>(N is number of positions shifted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRANCHING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>B</td>
<td>00111</td>
<td>400</td>
</tr>
<tr>
<td>Branch and Link</td>
<td>BAL</td>
<td>01011</td>
<td>400</td>
</tr>
<tr>
<td>Branch and Link Long</td>
<td>BALL</td>
<td>01010</td>
<td>800</td>
</tr>
<tr>
<td>Branch on Condition</td>
<td>BC</td>
<td>01000</td>
<td>400 - no branch 800 - branch</td>
</tr>
<tr>
<td>Skip on Condition</td>
<td>SKC</td>
<td>01001</td>
<td>400</td>
</tr>
<tr>
<td>Add to Storage and Skip</td>
<td>AS</td>
<td>01111</td>
<td>1200</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP</td>
<td>11111</td>
<td>400</td>
</tr>
</tbody>
</table>
RESULT TO REGISTER

Store to Register
Load from Register
Interchange Register

\[
\begin{array}{|c|c|c|}
\hline
\text{INSTRUCTION} & \text{MNEMONIC} & \text{OPERATION CODE} & \text{EXECUTION TIME IN NANOSECONDS} \\
\hline
\text{Store to Register} & \text{STR} & 1111 & 400 \\
\text{Load from Register} & \text{LR} & 1111 & 400 \\
\text{Interchange Register} & \text{IR} & 1111 & 400 \\
\hline
\end{array}
\]

Result Indicators

The SYSTEM/7 has six result indicators which show the status of a data operation result field. These result indicators are carry, overflow, result-zero, result-even, result-positive, and result-negative. Upon completion of most instructions, the resulting value is automatically tested for certain conditions, and the indicators are set accordingly. The discussion of the individual instructions, later in this chapter, gives details of when the indicators are set. Once an indicator is set, it will remain in that state until changed as the result of the execution of some instruction. The branch and state control instructions do not affect the result indicators.

The carry indicator is reset before each add, subtract, or non-circular left shift instruction is executed. If a one is shifted or carried out of bit zero by the execution of a left shift or add instruction the carry indicator is set. It
is also set when a subtraction operation attempts to borrow past bit zero.

The overflow indicator is set when the sign of the result of an arithmetic instruction is improper. The overflow indicator is reset before execution of an arithmetic instruction and each time the overflow indicator is tested.

The result-positive, result-negative, result-zero, and result-even indicators are reset before the execution of any instruction involving data manipulation (register or memory data). They are then set to reflect the resulting value obtained from the execution of the instruction.

The high-order bit of a word is the sign bit. A zero sign bit indicates a positive number which causes the result-positive indicator to be set. If the result field is zero the result-positive indicator is set off because the result-positive definition does not include a zero value.

A one in the sign bit indicates a negative number. Such a condition causes the result-negative indicator to be set. A negative zero is not represented in the SYSTEM/7. An operation result field has a value of zero if all its bits equal zero. Such a condition sets the result-zero indicator. An operation result field is even if bit-15 of the result field is zero. This causes the result-even indicator to be set.
SYSTEM/7 Instruction Set

In the following text, a summary of each instruction is given. The instruction opcode, name, and function is given for each instruction.

**Single-Word Operand Addressing Instructions**

The single-word instruction is 16 bits in length (Figure 4). The instruction is divided into several fields. The contents of these fields specify the operation to be performed and the location of the data to be processed. The Operation Code (opcode) field specifies which operation is to be performed. It is 5 bits in length (bits 0-4). The Register field (R) indicates whether the ACC, IAR, or one of the index registers is to take part in the instruction execution. The R field is 3 bits in length (bits 5-7). An R field of '000' specifies either the ACC or IAR, depending on the particular instruction. An R field of '001' through '111' specifies index registers 1 through 7, respectively. The contents of the register specified by the R field can be used for generating an effective address or as an instruction operand. The displacement field is 8 bits in length (bits 8-15).

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Name</td>
<td>Op Code</td>
<td>Register</td>
<td>Displacement</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4 - Short-Format Instruction Word**
The Effective Address (EA) is derived by algebraically adding the contents of the displacement field to the contents of the register specified by the R field. The ACC cannot be used in addressing.

The displacement can be either positive or negative. Therefore, the range of possible values in the displacement field is -128 to +127 (decimal).

The single-word operating instructions are discussed in the following text.

Opcode = 10001, A. -- The A instruction algebraically adds the contents of the ACC and the contents of the EA. The contents of the EA are unchanged. The carry, overflow, and result indicators reflect the contents of the ACC.

Opcode = 01111, AS. -- The AS instruction increments the location specified by the EA by one. The next sequential instruction (must be a one word instruction) is skipped if the result is zero; otherwise, program execution continues with the next sequential instruction. The result indicators reflect the contents of the EA.

Opcode = 00111, B. -- The B instruction performs an unconditional branch to the EA. No result indicators are effected.

Opcode = 01011, BAL. -- The BAL instruction places the contents of the IAR into the specified register (R) and places the EA into the IAR. No result indicators are effected.
Opcode = 11000, L. -- The L instruction loads the contents of the specified storage location into the ACC. The storage location is unchanged and the result indicators reflect the new contents of the accumulator.

 Opcode = 11001, LZ. -- The LZ instruction loads the contents of the EA into the ACC and sets the contents of the EA to zero. The result indicators reflect the new contents of the ACC.

 Opcode = 11100, N. -- The N instruction ANDs the contents of the specified storage location (EA) with the contents of the ACC. The result is placed in the ACC and the result indicators are set to reflect it.

 Opcode = 11101, 0. -- The O instruction ORs the contents of the EA with the contents of the ACC. The ACC gets the results of the operation and the result indicators are set to reflect the new ACC value.

 Opcode = 11010, ST. -- The ST instruction stores the contents of the ACC into the EA. The ACC is unchanged. The result indicators are set to reflect the new contents of the EA.

 Opcode = 01101, STX. -- The STX instruction causes the contents of the specified register (R) to be stored into the location specified by the EA. The contents of the register are unchanged. If the register field is zero, the
storage location is set to zero. (The STZ instruction is a mnemonic that will generate identical code.)

Opcode = 10010, S. -- The S instruction subtracts the contents of the EA from the ACC. The carry, overflow, and result indicators reflect the new contents of the ACC.

Opcode = 11110, X. -- The X instruction performs the exclusive OR of the contents of the EA and the contents of the ACC. The ACC receives the result and the result indicators are set to reflect it.

Non-Addressing Instructions

The non-addressing instructions are 16 bits in length (Figure 5). The Operation code and Register fields are identical to the corresponding fields for the single-word operand addressing instructions. The Modifier field is used in two ways. One use is for immediate data and the other modifies the opcode.

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>4 5 7 8</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>Register (R)</td>
<td>Modifier</td>
</tr>
</tbody>
</table>

Figure 5 - Non-Addressing Instruction Word

There are only two immediate instructions. A description of these follows.
Opcode = 01110, AI. -- The AI instruction algebraically adds the contents of the specified register (R) and the modifier (bits 8-15 with sign extended to 16 bits). Carry, overflow, and result indicators are set to reflect the new contents of the register (R).

Opcode = 01100, LI. -- The LI instruction loads the contents of the displacement field into the specified register. The sign bit (bit 8) of the displacement field is extended to the left forming a 16-bit word. The result indicators are set to reflect the new contents of the register.

The remainder of the non-addressing instructions have the same opcode (11111). The modifier field distinguishes what instruction is to be executed. Therefore, the following instructions in this class are identified below by their modifier instead of by their opcode.

Modifier = 0000 0001, AR. -- The AR instruction adds the contents of the register (R) and the ACC. The result is placed in the ACC. The contents of the register are unchanged, unless the ACC is specified for R. The carry, overflow, and result indicators are set to reflect the contents of the ACC.

Modifier = 0000 0010, SR. -- The SR instruction algebraically subtracts the contents of the register R from the ACC. The result is placed in the ACC. The carry, overflow, and result indicators reflect the new contents of the ACC.
Modifier = 0000 0011, NR. -- The NR instruction AND's the contents of the register R and the ACC. The result is placed in the ACC. The result indicators reflect the contents of the ACC.

Modifier = 0000 0100, OR. -- The OR instruction performs the logical OR of the register R and the ACC. The result is placed in the ACC. The result indicators are set to reflect the new contents of the ACC.

Modifier = 0000 0101, XR. -- The XR instruction performs the exclusive OR of the contents of the register R and the ACC. The result is placed in the ACC. The result indicators reflect the contents of the ACC.

Modifier = 0000 0110, STR. -- The STR instruction stores the contents of the ACC into the register R. The contents of the ACC are not changed. If zero is specified as the operand, the ACC contents are stored in the IAR. The result indicators reflect the contents of the register R.

Modifier = 0000 0111, LR. -- The LR instruction moves the contents of the register R into the ACC. If zero is specified as the operand, the IAR contents are moved into the ACC. Result indicators are set to reflect the contents of the ACC.
Modifier = 0000 1000, CR. -- The CR instruction performs the two's complement arithmetic on the contents of the register R. The overflow and result indicators reflect the result.

Modifier = 0000 1010, IR. -- The IR instruction interchanges the contents of the ACC and the register R. Result indicators reflect the contents of the ACC.

Shift Instructions

The shift instructions are 16 bits in length (Figure 6). The Operation code and Register fields are identical to the corresponding fields for the single-word operand addressing instructions. All shift instructions have the same opcode (00010), so the Modifier field distinguishes which instruction is being executed. The Count field is the shift count for the instruction.

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>11</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Name</td>
<td>Op Code</td>
<td>Register (R)</td>
<td>Modifier</td>
<td>Count</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6 - Shift Instruction Word

Since the shift instructions all have the same opcode, the modifier will be used to distinguish the various instructions in the following text.
Modifier = 000, SLC. -- The SLC instruction performs a circular left shift of the 16 bits in register R, by the number of bit positions specified in count. If R equals zero, the ACC is shifted. The result indicators reflect the new contents of the register R. A count operand of either zero or sixteen bits only sets the result indicators.

Modifier = 001, SLL. -- The SLL instruction performs a logical left shift (by the number of bits specified by Count) of the 16 bits in register R. Bits shifted off are lost, except the last one which sets the carry indicator. Vacated low-order positions are set to zero. Result indicators reflect the contents of the register R.

Modifier = 010, SRL. -- The SRL instruction performs a logical right shift (by the number of bits specified by Count) of the 16 bits in register R. Bits shifted off are lost. Vacated high-order positions are set to zero. Result indicators reflect new contents of register R.

Modifier = 011, SRA. -- The SRA instruction performs an arithmetic right shift (by the number of bits specified by Count) of the 16 bits in register R. Bits shifted off are lost. Vacated high-order bits are set to the value of the original sign bit (sign extension). Result indicators reflect new contents of Register R.
Double-Word Operand Addressing Instructions

The double-word operand addressing instructions are 32 bits in length (Figure 7). The Operation code and Register fields are identical to the corresponding fields for the single-word operand addressing instructions. R1 is the index register for computing the EA.

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>15</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Name</td>
<td>Op Code</td>
<td>Register (R)</td>
<td>R1</td>
<td>Not Used</td>
<td>Address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 7 - Double-Word Operand Addressing Instruction Word

The EA is computed differently for each of the double-word instructions. This will be explained in the text which follows.

Opcode = 10001, LXL. -- The LXL is one of the long-instructions. The contents of the EA are loaded into the specified register R. If R1 is zero, the EA is in the second word of the long-instruction. If R1 is non-zero, the EA is the algebraic sum of the address field (word two) of the long instruction and the contents of the index register specified by R1. Result registers are set to reflect the contents of the register specified by R.
Opcode = 01010, BALL. -- The BALL instruction moves the contents of the IAR into the register specified by R, and moves the Address (word 2) into the IAR. The result indicators are not affected.

**Conditional Skip**

The conditional skip instruction (SKC) is shown in Figure 8. The SKC instruction performs a conditional skip based upon the Condition Code (Figure 9). R is defined as before. If the Condition Code is satisfied a skip (one word) is performed. If not, no skip is performed. The overflow indicator is reset if tested; however, other program indicators are not affected. Extended mnemonics (Figure 10) simplify the coding of the SKC instruction. However, the Condition Code can be used if desired. Figure 9 lists the various condition codes and their meaning.
<table>
<thead>
<tr>
<th>Mask Bit</th>
<th>Operand Hex.</th>
<th>Machine Instruction Bit Position</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>X'80'</td>
<td>10</td>
<td>Result zero indicator on.</td>
</tr>
<tr>
<td>3</td>
<td>X'10'</td>
<td>11</td>
<td>Result negative indicator on</td>
</tr>
<tr>
<td>4</td>
<td>X'08'</td>
<td>12</td>
<td>Result positive indicator on</td>
</tr>
<tr>
<td>5</td>
<td>X'04'</td>
<td>13</td>
<td>Result even indicator on</td>
</tr>
<tr>
<td>6</td>
<td>X'02'</td>
<td>14</td>
<td>Carry indicator off</td>
</tr>
<tr>
<td>7</td>
<td>X'01'</td>
<td>15</td>
<td>Overflow indicator off</td>
</tr>
</tbody>
</table>

Figure 9 - Condition Code Masks

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Condition Code Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch on zero</td>
<td>BZ(R)</td>
<td>Bits 11 and 12 X'18'^</td>
</tr>
<tr>
<td>Branch on not zero</td>
<td>BNZ(R)</td>
<td>Bit 10 X'20'^</td>
</tr>
<tr>
<td>Branch on positive</td>
<td>BP(R)</td>
<td>Bits 10 and 11 X'30'^</td>
</tr>
<tr>
<td>Branch on not positive</td>
<td>BNP(R)</td>
<td>Bit 12 X'08'^</td>
</tr>
<tr>
<td>Branch on negative</td>
<td>BN(R)</td>
<td>Bits 10 and 12 X'28'^</td>
</tr>
<tr>
<td>Branch not negative</td>
<td>BNN(R)</td>
<td>Bit 11 X'10'^</td>
</tr>
<tr>
<td>Branch not even</td>
<td>BNE(R)</td>
<td>Bit 13 X'04'^</td>
</tr>
<tr>
<td>Branch on carry</td>
<td>BCY(R)</td>
<td>Bit 14 X'02'^</td>
</tr>
<tr>
<td>Branch on overflow</td>
<td>BO(R)</td>
<td>Bit 15 X'01'^</td>
</tr>
<tr>
<td>Unconditional branch</td>
<td>BL(R)</td>
<td>None X'00'^</td>
</tr>
</tbody>
</table>

Note: R is used for BCR instructions

<table>
<thead>
<tr>
<th>Skip on Condition</th>
<th>Mnemonic</th>
<th>Condition Code Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip on zero</td>
<td>SZ</td>
<td>Bit 10 X'20'^</td>
</tr>
<tr>
<td>Skip not zero</td>
<td>SNZ</td>
<td>Bits 11 and 12 X'18'^</td>
</tr>
<tr>
<td>Skip on positive</td>
<td>SP</td>
<td>Bit 12 X'08'^</td>
</tr>
<tr>
<td>Skip not positive</td>
<td>SNP</td>
<td>Bits 10 and 11 X'30'^</td>
</tr>
<tr>
<td>Skip on negative</td>
<td>SN</td>
<td>Bit 11 X'10'^</td>
</tr>
<tr>
<td>Skip not negative</td>
<td>SNN</td>
<td>Bits 10 and 12 X'28'^</td>
</tr>
<tr>
<td>Skip on even</td>
<td>SE</td>
<td>Bit 13 X'04'^</td>
</tr>
<tr>
<td>Skip no carry</td>
<td>SNC</td>
<td>Bit 14 X'02'^</td>
</tr>
<tr>
<td>Skip no overflow</td>
<td>SNO</td>
<td>Bit 15 X'01'^</td>
</tr>
</tbody>
</table>

Figure 10 - Extended Mnemonics
Conditional Branch

The conditional branch instruction (BC) is a two-word instruction. Figure 11 shows the format of the BC instruction.

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>4 5 7 8</th>
<th>15 16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Name</td>
<td>Op Code</td>
<td>Register (R)</td>
<td>Condition Bits</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 11 - Conditional Branch Instruction Format

The BC instruction performs a conditional branch based upon the condition bits (Figure 9). R is defined as before. A branch is made to the location specified by ADDRESS if no conditions are true. If any tested condition is true, program execution continues with the next sequential instruction. The overflow indicator is reset if tested; however, the other program indicators are not affected. Extended mnemonics (Figure 10) simplify the coding of the BC instruction. However, the condition code can be used if desired.

State Control Instructions

There are two State Control instructions. Each instruction is 16 bits in length and causes termination of execution in the simulator.
Opcode = 00110, LEX. -- The LEX instruction causes the processor to exit from the current priority level at which it is executing and enter the WAIT state. Since the SYSTEM/7 simulator only simulates one level of execution, the LEX constitutes an exit from the executing program. Program indicators are not affected.

Opcode = 00100, STP. -- The STP instruction causes the processor to enter the STOP state. This in effect terminates execution and causes the same results as the LEX instruction.
1. International Business Machines, IBM SYSTEM/7 Functional Characteristics, Form No. GA34-0003.
CHAPTER IV

LOADERS

Chapter IV discusses various loader schemes and presents the design for the SYSTEM/7 Loader.

Source decks are converted to object program decks by assemblers and compilers. The loader is a program which accepts an object program deck, prepares it for execution by the computer, and initiates the execution. The loader must perform four basic functions:

1. Allocate storage for the program (Allocation).
2. Resolve symbolic references (Linking).
3. Adjust address dependent locations (Relocation).
4. Place program into storage (Loading).

Chapter IV attempts to discuss the various loaders and how well they achieve the four basic functions.

Compile-and-Go Loaders

One method of performing the loader function is to have the assembler run in one section of storage and place the assembled instructions and data, as they are assembled, directly into their appropriate storage locations (Figure 12). When the assembly is completed, the assembler transfers control to the starting instruction of the program.
The Compile-and-Go scheme is relatively easy to implement; however, there are several disadvantages:

1. Storage is wasted, since the assembler must be storage resident.
2. The program must be reassembled each time it is run.

![Figure 12 - Compile-and-Go Loader Scheme](image-url)
General Loader Scheme

Outputing assembled instructions and data as they are assembled alleviates the problem of wasting storage for the assembler. Such output can be saved and loaded whenever the code is to be executed. This allows the assembled program to use storage previously occupied by the assembler. This form of output is called an object module.

The use of an object module as intermediate data necessitates the introduction of a loader (Figure 13). The loader accepts the object module, and places machine instructions and data into storage in an executable format. The loader is assumed to be much smaller than the assembler, so that more storage is available to the user. Another advantage is that reassembly is not necessary each time a program is run. Also, if all source program translators (Assembler and Compilers) produce compatible object modules, it is possible to write subroutines in various languages and link them together through the loader. Libraries of existing routines could be linked also.

Source program Translator Object program Loader

Source program Translator Object program

Figure 13 - General Loader Scheme
Absolute Loaders

The simplest type of loader scheme, which fits the general scheme shown in Figure 13, is the absolute loader. In this scheme, the assembler outputs an object module in almost the same form as the Assemble-and-Go scheme, except data is not placed directly into storage by the assembler. The loader accepts the object module and places it into storage at the location specified by the assembler.

Absolute loaders (1) are very simple to implement; however, there are several disadvantages:

1. The programmer must specify the address at which the program will be loaded.
2. For multiple subroutines the programmer must perform his own linking.
3. For multiple subroutines, a change in one subroutine requires that all routines be reassembled.

Relocating Loaders

The relocating loader was developed to avoid having to reassemble all subroutines in a program, when a single subroutine is changed, and to perform the tasks of allocation and linking for the programmer. Using relocating loaders, the assembler assembles each procedure segment independently and passes on to the loader the information relative to relocation and intersegment references.
In earlier computers, programs were punched on cards or tape and loaded in at the beginning of storage; however, with relocatability came the capability of loading a program anywhere outside the storage range of the control program (5).

Direct Linking Loaders

The direct linking loader is a general relocatable loader. It is perhaps the most popular loading scheme presently in use. The direct linking loader is often called a linkage editor. This type of loader analyzes the source programs to be executed and specifies the absolute address for data and symbolic names. Since the location where the program is to be loaded into storage is not known, this loader scheme operates on the principle of taking a symbolic address and giving it a displacement from the beginning of the load module. IBM uses this loader scheme (4).

The direct linking loader allows the programmer to use multiple procedure segments and multiple data segments. It also gives him complete freedom to reference data or instructions located in other segments. This provides flexible intersegment referencing while allowing independent translation of programs.

The formats of assembler output may vary, but the information furnished to the loader is basically the same.
The relocating assembler must provide the loader with the following information:

1. Segment length.
2. List of all entry points and their relative location.
3. Information regarding location and modification of address constants.

**SYSTEM/7 Loader**

The SYSTEM/7 Simulator uses the standard IBM SYSTEM/7 assembler (2) and linkage editor (3) before simulation execution begins. The SYSTEM/7 linkage editor (Link/7) is a direct linking loader.

Link/7 processes load modules and object modules produced by SYSTEM/7 programs. The modules are the output of the SYSTEM/7 assembler (ASM/7). Object and load modules must be processed by a formatting program (Format/7) (2) before execution. Figure 14 illustrates the processing of a SYSTEM/7 program with no external references. Figure 15 illustrates the processing of a program which references external modules.

A program written in assembler language is known as a source module. The assembler processes the source module and produces an object module in machine language. The assembler outputs a listing of the assembled program.
Figure 14 - Processing a SYSTEM/7 Program Which Does Not Require Linkage with Other Modules
Figure 15 - Processing SYSTEM/7 Programs Using the Linkage Editor
The assembler produces four types of cards in the object deck: ESD, TXT, RLD, and END. External Symbol Dictionary (ESD) cards contain information about all symbols that are defined in the program that may be referenced elsewhere, and all symbols defined as externals in the program.

The Text (TXT) cards contain the object code version of the source program. Each text record contains the origin address of the instructions or data included in the record.

The Relocation and Linkage Dictionary (RLD) cards contain information about those locations in the program whose contents must be modified due to relocation. For such locations the loader must supply information enabling the loader to correct their contents.

The END card indicates the end of the object deck to Link/7 and specifies the starting address for execution if the assembled routine is the main program. This record may also contain the control section length if it was not previously specified in the ESD SD item.

Figure 16 illustrates the way an object deck would appear. The exact format and contents of each type of card

![Diagram of ESD, TXT, RLD, END cards]

Figure 16 - Example of Object Deck for Link/7
in the object deck can be found in the IBM SYSTEM/7 Linkage Editor (2) manual.

After the linking process is complete, the object module must be formatted before it can be executed. This is accomplished by the IBM Format/7 routines. The Format/7 utility program reformats ASM/7 object modules or Link/7 load modules into storage modules that can be loaded into storage for execution.

Format/7 produces two types of output records: TEXT and END. A TEXT record contains data that will be loaded into storage starting at a specified address. The END record contains the starting address of the program. The format and contents of the TEXT and END cards are shown here, since this is the input to SIM/7.

**'TX' Record**

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Usage/Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>ID</td>
</tr>
<tr>
<td>3-4</td>
<td>Type (TX)</td>
</tr>
<tr>
<td>5-6</td>
<td>Data Word Count</td>
</tr>
<tr>
<td>7-8</td>
<td>Data or Entry Point Address</td>
</tr>
<tr>
<td>9-80</td>
<td>Object Code to be Loaded at Address in Bytes 7-8</td>
</tr>
</tbody>
</table>

**'EN' Record**

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Usage/Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>ID</td>
</tr>
<tr>
<td>3-4</td>
<td>Type (EN)</td>
</tr>
<tr>
<td>5-6</td>
<td>Data Word Count (Always 1)</td>
</tr>
<tr>
<td>7-8</td>
<td>Entry Point Address</td>
</tr>
</tbody>
</table>
The storage module output by Format/7 is input to the bootstrap loader of the SYSTEM/7 Simulator. The storage module is a binary deck in the format previously shown. The binary deck is decoded using the PL/I function UNSPEC. The data and instructions are then loaded into the simulated storage at the location specified on each text card. When an END card is detected, the loader branches to the specified location and the simulator begins execution of the program. Figure 17 illustrates the flow of execution of the loader for SIM/7.
Figure 17 - Flow Diagram of Loader for SYSTEM/7 Simulator

2. International Business Machines, IBM SYSTEM/7 Linkage Editors, Form No. GC34-0006.

3. International Business Machines, IBM SYSTEM/7 Macro Assemblers, Form No. GC34-0018.

4. International Business Machines, Linkage Editor and Loader, Form No. GC28-6538.

CHAPTER V

THE DEFINITION OF SIM/7 USING THE VIENNA DEFINITION LANGUAGE

Introduction

The Vienna Definition Language (VDL) was designed for defining other programming languages (3). However, in this chapter VDL will be used to define the processor module for the IBM SYSTEM/7 SIMULATOR. First, a brief introduction to the basis definition techniques of VDL is presented. Then the definition of the processor module for the IBM SYSTEM/7 Simulator is developed.

The Vienna Definition Language

VDL has been described as a meta language for defining interpreters. It was inspired by McCarthy's interpreter-oriented definition of LISP, his subsequent definition of programming language interpreters in terms of state vector transformations, and his definitions of interpreters such as the SECD machine (3). There is, however, one big advance in the concepts of the Vienna approach to language definition that is not present in earlier interpreter-oriented definitions. That is the notion that both the syntax of programs and the syntax of complete states are described in
terms of the same syntactic notation. VDL, in using the same syntactic notation to describe program structures and execution-time data structures, uses a tree structure. The class of Vienna objects has been deliberately restricted to tree structures because it has been determined that tree structures are adequate for the representation of programs and structured states (2).

In VDL, states that may occur during the execution of a program are described in terms of an underlying class of data structures. State transformations corresponding to the execution of instructions of the defined programming language are described in terms of transformation operators applicable to data structures that represent states.

There are two classes of data objects in the Vienna Definition Language:

1. SIMPLE OBJECTS - elements of this class have no components but have data structure transformation attributes when they occur as operators or operands during program execution.

2. COMPOSITE OBJECTS - composite objects are built from elementary objects by construction operators. Elements of this class have components that may be selected by unique selectors. The components may be either elementary objects or composite objects.
The Data Set

A tree is represented by a set of pairs where each pair describes a single branch and consists of a selector and an object where the selector is equivalent to a pointer to the objects and represents the name of the branch of the tree leading to the object. A named object is described by a pair \( \{ s:A \} \) where \( s \) is the selector of the object \( A \). The simplest possible data element is the tree which contains one branch and one object as shown in Figure 18.

![Figure 18 - A Simple Object](image)

This tree is characterized by the set containing one pair \( \{ s:A \} \). For purposes of description, it is necessary to ascribe names to trees. For this purpose we shall use the symbol '=' , which will be interpreted as the naming operator. Thus, if the tree shown in Figure 18 were given the name \( B \), we would write the following:

\[ B = \{ s:A \} \]
Operations Over the Data Set

The application of a single selector to an object is said to yield its immediate component. Thus, given the object $B$, we may select its component by applying the selector(s). When an object is composed of several branches, the selector that is applied to the object determines which of several choices of immediate components is to be obtained. For example, the application of selector $s_3$ to the object named $B$ in Figure 19 will yield the component $A_3$.

![Figure 19 - A Composite Object](image)

Since the application of a selector to an object yields another object, the result may also be subject to the application of a selector. The successive application of selectors to an object can be nested as follows:

$$s_1(s_2(s_3(s_4(B))))$$

Such notation is clumsy and dangerous. Thus the notation

$$s_1 \circ s_2 \circ s_3 \circ s_4(B)$$

is utilized.
When the object to which a selector is applied had no edge labeled "s" emanating from its root vertex, then \( s(t) \) is defined as the null object \( \emptyset \).

The Mutation Operator

The operations of tree walking, which are accomplished by means of the selectors, are not sufficient to perform operations of mutations on the data elements. Thus, a new operation must be introduced with the purpose of performing changes to the data trees. The fundamental operation of mutation is defined as follows:

\[
(A; \langle s:B \rangle )
\]

where \( A \) is the characteristic set of the object to be mutated and \( \langle s:B \rangle \) is a named object to be added to the copy of the object \( A \), and where \( s \) is a simple selector. For example, given the object

\[
z = \{\langle s1:e1 \rangle , \langle s4: \{\langle s2:e2 \rangle , \langle s3:e3 \rangle \} \} \}
\]

as shown in Figure 20, the creation of a new object \( y \), by the operation

\[
Y = \cup (Z; \langle s5:e5 \rangle )
\]

would result in an object with the characteristic set

\[
\{\langle s1:e1 \rangle , \langle s4: \{\langle s2:e2 \rangle , \langle s3:e3 \rangle \} \}, \langle a5:e5 \rangle \}
\]
Similarly the operation

$$X = \mathcal{M}(Z; \langle s4:e4 \rangle)$$

would result in the characteristic set

$$\{ \langle s1:e1 \rangle, \langle s4:e4 \rangle \}$$

as shown in Figure 20.

![Diagram showing objects Z, Y, and X with characteristics](image)

**Figure 20 - The Mutation of an Object**

**Relationships Between Objects**

Control options are needed to choose a particular set of further operations to be performed, based on the current state of the objects in the system. To be able to make such decisions, a set of functions must be provided whose range
is the set of truth values \( \{T, F\} \) over the domain of objects. The relational functions \( (=, \neq, \geq, >, \leq, <) \) is such a set.

Besides requiring that the equality or inequality of objects be tested, it is also convenient to provide a set of predicates which are defined over the domain of some prescribed objects and the whole universe of objects in the system with the range of truth values \( \{T, F\} \). Actually a predicate indicates equality of two objects or membership of an object in a specified class of objects. In its simplest form, a predicate in the domain of elementary objects determines the equality of two elementary objects. For the purpose of identification, all predicates are syntactically recognized by the prefix is-, where the suffix specifies the object to be used as the base comparator.

The inequality of objects may be expressed by using the logical negation operator "\( \neg \)". or by the prefix "not-". The logical exclusive-OR operator can be expressed by the symbol "\( \lor \)".

Instruction Execution

The semantics of a programming language is defined in VDL in terms of the sequences of information structure transformations to which its programs give rise during execution (1). Every computation starts with an initial state.
A state is represented in VDL as a composite object whose immediate components $s$ may be selected by selectors $s$ and whose lower-level components may be selected by sequences of selectors. Both data and instructions may be regarded as part of the state $\xi$.

At any given point of execution, the instructions of a VDL program are represented by a tree structure called a control tree. Figure 21a illustrates a control tree containing six instructions. Figure 21b illustrates how the control tree can be represented by identification.

(a) Tree Structure  (b) Indented Structure

Figure 21 - The Structure of Control Trees

The instruction execution cycle of VDL specifies that, upon completion of an instruction, any terminal vertex of a control tree may be executed as the next instruction. Thus, in Figure 21, any one of the four instructions Instr2, Instr5, Instr6, and Instr4 may be executed next.
Instructions may be defined as being interpretable in either of two manners:

1. Self-Replacing (macro) Instructions - that is, the definition is a set of instructions which are to replace the instruction being interpreted, or

2. Value-Returning Instructions - that is, instructions which affect the data part of the state and which provide values to the argument lists of other instructions which are to be executed in some later state.

Value-Returning Instructions have the following general format:

\[
\text{PASS} \leftarrow e_0 \\
s-sc_1 \leftarrow e_1 \\
s-sc_2 \leftarrow e_2 \\
\vdots \\
s-sc_n \leftarrow e_n
\]

Evaluate the expressions \(e_0, e_1, e_2, \ldots, e_n\). Pass the value \(e_0\) to predecessor vertices to be stored as the value returned by the instruction. Assign the value of \(e_i\) to the state component \(s-sc_i\) for \(i=1,2,\ldots, n\).

If Instr6, Figure 21 is a value returning instruction with the above format, then its execution would cause the following:

1. the vertex Instr6 would be deleted from the control tree.
2. the instruction parameters of the predecessor vertices of the control tree would be assigned the value of the expression $e_0$.

3. each of the state components $s_{sci}$, $i=1,2,...,n$, would be updated to the value of the corresponding expression $e_i$.

If, on the other hand, the instruction were a macro instruction, its execution would cause the control tree vertex $Instr_6$ to be replaced by an instruction subtree.

The semantics of VDL will be illustrated in the next section by defining the processor module for the IBM SYSTEM/7 Simulator.

A VDL DESCRIPTION OF THE IBM SYSTEM/7 SIMULATOR

Since the intention of a definition language is to describe formally, accurately, concisely and unambiguously processes which are delivered to a computer for "execution," then the method of definition must be capable of describing a processor itself. Since the execution of instructions within a processor causes side effects on the registers of the processor, it is particularly important to a designer to have a detailed definition when considering the addition of new features to that processor. This is particularly important in the modification of a simulator.
A simplified block diagram of the processor for the SYSTEM/7 Simulator is shown in Figure 22. Initially the structure of the state of the machine is defined to contain the components which represent the major registers of the machine:

\[
\text{is-\mathcal{C}} = (\langle \text{s-acc:is-WORD} \rangle, \langle \text{s iar:is-WORD} \rangle, \langle \text{s-sar:is-WORD} \rangle, \langle \text{s-sdr:is-WORD} \rangle, \langle \text{s-op:is-WORD} \rangle, \langle \text{s-xr(1):is-WORD} \rangle, \langle \text{s-xr(2):is-WORD} \rangle, \langle \text{s-xr(3):is-WORD} \rangle, \langle \text{s-xr(4):is-WORD} \rangle, \langle \text{s-xr(5):is-WORD} \rangle, \langle \text{s-xr(6):is-WORD} \rangle, \langle \text{s-xr(7):is-WORD} \rangle, \ldots )
\]

where additional components will be added later.

Representations of Registers

Obviously, the basic element of a register is a single bit of data which we shall consider to be an elementary object of the definition system. That is, the object which conforms to the predicate is-bit, also conforms to the predicate is-e0. On this basis, we may represent a word in the memory of the processor as a list of objects which conform to the predicate is-bit. That is,

\[
is-\text{WORD}=\text{is-bit-list}
\]

where

\[
\text{bit}(i) \equiv \text{elem}(i \ 1)
\]
Figure 22 - Processor Data Flow
Additionally we shall define a special length function:

\[ \text{word-length}(A) = \begin{cases} 
\text{is-bit-list}(A) \rightarrow \text{length}(A) - 1 
\end{cases} \]

Also, assuming that the 0-th bit in a word represents the high order bit of the number and the word-length bit contains the low order bit, then we shall define the function:

\[ \text{convert}(A) = \text{is-bit-list}(A) \rightarrow \begin{cases} 
\text{is-}<>^0(A) \rightarrow 0, \\
T \rightarrow \text{head}(A) \times 10 \text{ (word-length}(A) + \text{convert}(\text{tail}(A)) \end{cases} \]

Using these definitions we may now expand on the abstract structure of the machine:

\begin{align*}
\text{is-WORD} &= (\textless \text{bit}(i) : \text{is-bit} \rangle \mid 0 \leq i \leq 1111) \\
\text{is-OP} &= (\textless \text{bit}(i) : \text{is-bit} \rangle \mid 0 \leq i \leq 0100) \\
\text{is-IAR} &= (\textless \text{s-WORD} : \text{is-word-address} \rangle)
\end{align*}

The component of the state of the machine which represents memory can be defined as follows:

\[ \text{is-}薛 = (\ldots, \\
\text{s-mem} : \text{is-mem}, \\
\ldots) \]

where

\[ \text{is-mem} = (\{ \textless \text{s-WORD}(i) : \text{is-WORD} \rangle \mid 0 \leq i \leq 1000000000000 \}) \]

and hence in order to address the words of memory,

\[ \text{is-word-address} = (\{ \textless \text{bit}(i) : \text{is-bit} \rangle \mid 0 \leq i \leq 1111 \}) \]
Register Functions

The functions of the major registers in the SYSTEM/7 Simulator are to store, control, or retrieve data in the system, to modify the data in the memory and to prepare instructions fetched from memory for interpretation.

The Storage Data Register (SDR) acts as the intermediate storage register for interregister data transfers. Every word of data transferred to or from main storage by the processor passes through the SDR.

The Accumulator (ACC or XR(O)) is used in most arithmetic and logical operations. Such operations are performed using one specified operand (from storage or from an index register) and one implied operand (the value previously loaded into the accumulator). The result is in the accumulator at the end of the operation.

The Storage Address Register (SAR) is a 16-bit register used by the processor to access each location in main storage. The contents of the SAR cannot be controlled by the programmer.

The Instruction Address Register (IAR) holds the address of the next instruction to be executed. During program execution the contents of the IAR are placed in the SAR for fetching the contents of the next location in storage. The IAR is incremented immediately after its contents have been placed in the SAR. Consequently, the IAR contains the
address of the next location in storage at the time the current instruction is executed. Sometimes, however, the contents of the IAR are changed as a result of the instruction being executed (the branch instruction is an example of this).

The Operation Register (OP) is a 16-bit register that holds the first 16 bits of the instruction accessed from storage. In most cases, these 16 bits are the complete instruction; however, the instruction could be 32 bits in length (two word 'long' instruction).

The Index Registers (XR(1) to XR(7)) are 16-bit registers. The contents of the Index Registers can be used in generating the effective address (EA) of an instruction. The contents of Index Registers can also be manipulated in arithmetic and logical operations.

The basic machine, without peripheral input/output devices, has only three significant states. These three states are Fetch, Effective-Address-Generation, and Execute. The Fetch state is the first entered for the interpretation of any instruction. During this phase the IAR is moved into the SAR, the IAR is incremented, and the instruction is retrieved from memory, from the location designated by the SAR, and is placed in the SDR.
Memory referencing instructions may cause the activation of the Effective-Address-Generation state. Otherwise, the execute state is entered directly to actually execute the instruction.

Fetch State

The following components will now be added to the system:

\[
is-\mathcal{G} = \left\{ \ldots \right.\]
\[
\left. \langle s\text{-cnst}:i\text{-WORD} \rangle, \right.
\left. \langle s\text{-zro}:i\text{-ZRO} \rangle, \right.
\left. \langle s\text{-ones}:i\text{-ONES} \rangle, \right.
\left. \langle s\text{-ea}:i\text{-WORD} \rangle, \right.
\left. \langle s\text{-rslt}:i\text{-RSLT} \rangle, \right.\]
\[
\ldots \}
\]

where

\[
is\text{-ZRO} =
\left\langle 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 \right\rangle
\]

\[
is\text{-ONES} =
\left\langle 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 \right\rangle
\]

\[
is\text{-RSLT} =
\left( \{ \langle \text{bit}(i):i\text{-BIT} \rangle \mid 0 \leq i \leq 0110 \} \right)
\]

The relations \( s\text{dr-xr}(j) \), \( \text{mem-sdr} \), and \( \text{set-result} \) will also be defined here.

\[
s\text{dr-xr}(j) =
\left( s\text{-xr}(j):s\text{-word}o\text{-s-dr}(\mathcal{G}) \right)
\]

where

\[
J = \text{Convert} \left( \{ \langle \text{bit}(i,s\text{-s-dr}(\mathcal{G})) \mid 101 \leq i \leq 111 \} \right)
\]

(Note: \( \text{xr}(0) \equiv \text{acc} \))

\[
\text{mem-sdr} =
\left( s\text{-s-dr}:s\text{-word}o\text{-s-mem}(\mathcal{G}) \right)
\]

\[
\text{set-result}(x) =
\left( s\text{-x} = \langle 0000000000000000 \rangle \rightarrow \text{set-zdnp0} \right)
\]

\[
T \rightarrow \text{set-zenp}
\]
where

\[
\text{set-zenp0} = \\
\{ \text{bit}(i)^0 \text{s-rslt}: 1 \mid i = 0, 3 \} \\
\{ \text{bit}(i)^0 \text{s-rslt}: 0 \mid i = 1, 2 \}
\]

and

\[
\text{set-zenp} = \\
\text{set-z} \\
\text{set-e} \\
\text{set-np}
\]

where

\[
\text{set-z} = \text{bit}(0)^0 \text{s-rslt}: 0
\]

and

\[
\text{set-e} = \\
\text{bit}(1111, s-x) = 1 \rightarrow \text{bit}(3)^0 \text{s-rslt}(\_): 0 \\
T \rightarrow \text{bit}(3)^0 \text{s-rslt}(\_): 1
\]

and

\[
\text{set-np} = \\
\text{bit}(0, s-x) = 1 \rightarrow \text{bit}(2)^0 \text{s-rslt}(\_): 0 \\
\text{bit}(1)^0 \text{s-rslt}(\_): 1 \\
T \rightarrow \text{bit}(2)^0 \text{s-rslt}(\_): 1 \\
\text{bit}(1)^0 \text{s-rslt}(\_): 0
\]

The Fetch state can be defined as follows:

\[
\text{FETCH} = \\
\text{s-run}(\_)=1 \rightarrow \\
\text{op-eped}; \\
\text{sdr-ep}; \\
\text{mem-sdr}; \\
\text{inc-iar}; \\
\text{iar-sar}; \\
T \rightarrow \text{NULL}
\]

Transfer of the contents of the IAR to the SAR can be des-
scribed as follows:

\[
\text{iar-sar} \\
\text{s-sar:s-word}^0 \text{s-iar}(\_)
\]
Following the moving of the IAR to the SAR the IAR is incremented. The precise technique for incrementation of the IAR is not specified in the SYSTEM/7 Functional Characteristics manual (5), thus, the instruction inc-iar will be described as being implementation-defined. Now data must be moved from memory to the SDR which is driven by the SAR.

\[
\text{mem-sdr} = \text{s-sdr:s-word(convert( } s\text{-word}^0\text{-s-sar}(<>) ))
\]

Next the instruction is copied from the SDR into the OP.

\[
\text{sdr-op} = \text{s-op:s-word}^0\text{s-sdr}(<>)
\]

Now we must select the operation code from the instruction OP.

\[
\text{op-opcd} = \text{s-opcd: } \{ \text{bit}(k,s\text{-op}(<>) ) \mid 0 \leq k \leq 100 \}
\]

Having selected the Operation code from the instruction contained in OP we are now ready to move into the execute state. The Effective-Address-Generation state is entered during the Execute state only if the instruction being executed requires that an effective address be calculated.

Effective-Address-Generation State

The Effective-Address-Generation state can be defined as follows:

\[
\text{create-ea} = \text{Convert( } \{ \text{bit}(i,s\text{-sdr}(<>) ) \mid 0 \leq i \leq 100 \} ) =
\]

\[
\rightarrow \text{create-lginst}
\]

\[
\rightarrow \text{create-brdisp}
\]
The op codes 10001, 01000, and 01010 represent the three long instructions, in the SYSTEM/7 Simulator, which require effective address generation. In long instructions the second word of the instruction is the effective address. Therefore, the calculation of the effective address for a long instruction can be defined as follows:

\[
\text{create-1ginst} = \text{s-word(convert(s-word's-sar(\_)))}
\]

The SYSTEM/7 operates on a base-register-displacement principle for storage addressing. In a short-format instruction, the effective address is derived by adding algebraically the contents of the displacement field (bits 8-15) to the contents of the register indicated by the R field (bits 5-7). The R field in the instruction specifies that the contents of the IAR(R=000) or one of the index registers (R=001 to 111) are added to the displacement value to compute the effective address. The displacement can be defined as follows:

\[
\text{create-disp} =
\begin{cases}
\text{bit(10000, s-sdr(\_)))} = 0 \\
\text{s-disp: } <0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0>
\end{cases}
\]

The base register can be defined as follows:

\[
\text{create-reg} =
\begin{cases}
\text{convert(bit(1,sdr(\_))) } | 101 \leq i \leq 111 \\
\text{xr(convert(bit(1,sdr(\_))) } | 101 \leq i \leq 111)
\end{cases}
\]
Now the base-register-displacement address can be defined as follows:

\[
\text{create-brdisp} = \\
\text{sum}(\text{create-reg}, \text{create-disp}) \\
\text{create-reg} \\
\text{create-disp}
\]

where \text{sum}(x,y) is defined to be the algebraical sum of \(x\) and \(y\).

The Execute State

Having determined the method for creating the effective address for memory referencing instructions, we may now define the various instructions.

LOAD (L)

\[
\text{execute-load} = \\
\text{set-result} \\
\text{sdr-acc} \\
\text{mem-sdr} \\
\text{create-ea}
\]

LOAD AND ZERO (LZ)

\[
\text{execute-lz} = \\
\text{set-result} \\
\text{sdr-mem} \\
\text{reset-sdr} \\
\text{sdr-acc} \\
\text{mem-sdr} \\
\text{create-ea}
\]

where

\[
\text{reset-sdr} = \\
\text{s-sdr: } <0,0,0,0,0,0,0,0,0,0,0,0,0,0>
\]
LOAD IMMEDIATE (LI)

execute-li=
  set-result
  cnst-xr(j)

where

\[ cnst-xr(j) = \begin{cases} 
  & \text{bit}(1000, s - \text{sdr}(\mathbf{x})) = 0 \rightarrow \\
  & s-xr(j): \langle 0, 0, 0, 0, 0, 0, 0, 0 \rangle \bigwedge \{ \text{bit}(i, s - \text{sdr}(\mathbf{x})) \} \mid 1000 \leq i \leq 1111 \\
  & T \rightarrow s-xr(j): \langle 1, 1, 1, 1, 1, 1, 1, 1 \rangle \bigwedge \{ \text{bit}(i, s - \text{sdr}(\mathbf{x})) \} \mid 1000 \leq i \leq 1111 
\end{cases} \]

LOAD INDEX REGISTER LONG (LXL)

execute-lxl=
  set-result
  sdr-xr(j)
  mem-sdr
  create-ea

STORE (ST)

execute-st=
  set-result
  sdr-acc
  mem-sdr
  create-ea

STORE INDEX (STX)

execute-stx=
  set-result
  sdr-mem
  xr(j)-sdr
  create-ea

ADD (A)

execute-a=
  set-result
  sum(sdr, acc)
  mem-sdr
  create-ea
ADD IMMEDIATE (AI)

execute-ai=
    set-result
    sum(cnst,xr(j))
    sdr-cnst

where

sdr-cnst=
    bit(1000,s-sdr(\$))=0 \rightarrow
    s-cnst: \langle 0,0,0,0,0,0,0,0,0,0 \rangle \rightarrow \{\langle bit(i,s-sdr(\$)) \rangle | 1000 \leq i \leq 1111\}
    T\rightarrow s-cnst: \langle 1,1,1,1,1,1,1,1,1,1 \rangle \rightarrow \{\langle bit(i,s-sdr(\$)) \rangle | 1000 \leq i \leq 1111\}

SUBTRACT (S)

execute-s=
    set-result
    sub(sdr,acc)
    mem-sdr
    create-ea

where sub(x,y) is defined to be the algebraic difference of x and y (x-y), with the result residing in the ACC.

ADD REGISTER (AR)

execute-ar=
    set-result
    sum(xr(j),acc)

SUBTRACT REGISTER (SR)

execute-sr=
    set-result
    sub(xr(j),acc)

COMPLEMENT REGISTER (CR)

execute-cr=
    set-result
    comp-i

where

comp-i=
    bit(i,s(xr(j)))=1 \rightarrow bit(i)\circ s-xr(j):0
    T\rightarrow bit(i)\circ s-xr(j):1
AND (N)

execute-n=
set-result
  sdr-and-acc
  mem-sdr
  create-ea

where

  sdr-and-acc=
  and-0
  and-1
  ...
  and-1111

where

  and-i=
  bit(i,s-acc)=1 \rightarrow \text{bit(i)}^0 \text{s-acc}:\text{bit(i,s-sdr(\text{\$}))}
  T \rightarrow \text{NULL}

OR (O)

execute-o=
set-result
  sdr-o-acc
  mem-sdr
  create-ea

where

  sdr-o-acc=
  o-0
  o-1
  ...
  o-1111

where

  o-i=\text{bit(i,s-acc)=1} \rightarrow \text{bit(i)}^0 \text{s-acc}:1
  T \rightarrow \text{bit(i)}^0 \text{s-acc}:\text{bit(i,s-sdr(\text{\$}))}

EXCLUSIVE OR (X)

execute-x=
set-result
  sdr-x-acc
  mem-sdr
  create-ea
where

\[ sdr-x-acc = \]
\[ x-0 \]
\[ x-1 \]
\[ \ldots \]
\[ x-1111 \]

where

\[ x-i = \]
\[ bit(i,s-acc) = 1 \rightarrow bit(i) \circ s-acc : bit(i,s-sdr(\text{E})) \]
\[ T \rightarrow bit(i) \circ s-acc : bit(i,s-sdr(\text{E})) \]

AND REGISTER (NR)

execute-nr=
  set-result
  \[ xr(j)-nr-acc \]

where

\[ xr(j)-nr-acc = \]
\[ nr-0 \]
\[ nr-1 \]
\[ \ldots \]
\[ nr-1111 \]

where

\[ nr-i = bit(i,s-acc) = 1 \rightarrow bit(i) \circ s-acc : bit(i,s-xr(j)) \]
\[ T \rightarrow \text{null} \]

OR REGISTER (OR)

execute-or=
  set-result
  \[ xr(j)-or-acc \]

where

\[ xr(j)-or-acc = \]
\[ or-0 \]
\[ or-1 \]
\[ \ldots \]
\[ or-1111 \]

where

\[ or-i = bit(i,s-acc) = 1 \rightarrow bit(i) \circ s-acc : 1 \]
\[ T \rightarrow bit(i) \circ s-acc : bit(i,s-xr(j)) \]
EXCLUSIVE OR REGISTER (XR)

execute-xr=
  set-result
  xr(j)-xr-acc

where

xr(j)-xr-acc=
xr-0
xr-1
...
xr-1111

where

xr-i=bit(i,s-acc)=1 → bit(i)=s-acc:bit(i,s-xr(j))
T → bit(i)=s-acc:bit(i,s-xr(j))

SHIFT LEFT CIRCULAR (SLC)

execute-slc=
  set-result
  slc-xr(j)

where

slc-xr(j)= \{<\text{bit}(i,s-xr(j))> | cnt+1 \leq i \leq 1111\}
\{<\text{bit}(i,s-xr(j))> | 0 \leq i \leq cnt\}

SHIFT RIGHT LOGICAL (SRL)

execute-srl=
  set-result
  srl-xr(j)

where

srl-xr(j)= \{<\text{bit}(i,s-zro)> | 0 \leq i \leq \text{cnt}+1\}
\{<\text{bit}(i,s-xr(j))> | 0 \leq i \leq 1111-\text{cnt}\}

SHIFT RIGHT ARITHMETIC (SRA)

execute-sra=
  set-result
  sra-xr(j)
where

\[ sra-xr(j) \quad \text{bit}(0, xr(j)) = 1 \rightarrow \]
\[ \{ \text{bit}(i, s-ones) \mid 0 \leq i \leq \text{cnt+1} \} \]
\[ \{ \text{bit}(i, s-xr(j)) \mid 0 \leq i \leq 1111-cnt \} \]
\[ \text{branch} (B) \]
\[ \text{execute-b}= \]
\[ \text{ea-iar} \]
\[ \text{create-ea} \]

where

\[ \text{ea-iar} = s \text{-iar}: s \text{-word s-ea(\$)} \]

\[ \text{branch and link} \ (\text{BAL}) \]
\[ \text{execute-bal}= \]
\[ \text{ea-xr(j)} \]
\[ \text{create-ea} \]

where

\[ \text{ea-xr(j)}: s \text{-word s-ea(\$)} \]

\[ \text{branch and link long} \ (\text{BALL}) \]
\[ \text{execute-ball}= \]
\[ \text{ea-xr(j)} \]
\[ \text{create-ea} \]

\[ \text{conditional branch} \ (\text{BC}) \]
\[ \text{execute-bc}= \]
\[ \{ \text{bit}(i, s-sdr) \mid 1010 \leq i \leq 1111 \} = 0 \rightarrow \text{ea-iar} \]
\[ \text{test-result} \]
\[ \rightarrow \text{ea-iar} \]
\[ \text{branch} \ (\text{NULL}) \]

\[ \text{test-result}= \]
\[ \{ \text{bit}(i, s-sdr) \mid 1010 \leq i \leq 1111 \} \quad \text{and.} \]
\[ \{ \text{bit}(i, s-rslt) \mid 0 \leq i \leq 0110 \} = .T. \]
\[ \rightarrow \text{ea-iar} \]
\[ \text{branch} \ (\text{NULL}) \]
SKIP CONDITIONAL (SKC)

\[
\text{execute-skc} = \begin{cases} \langle \text{bit}(i, s-sdr) \rangle & \text{if } 1010 < i \leq 1111 \text{ and } \langle \text{bit}(i, s-sdr) \rangle = 0 \rightarrow \text{inc-iar} \\ T \rightarrow \text{test-result} \end{cases}
\]

where

\[
\text{test-result} = \begin{cases} \langle \text{bit}(i, s-sdr) \rangle & \text{if } 1010 < i \leq 1111 \\ \langle \text{bit}(i, s-srlt) \rangle & \text{if } 0 \leq i \leq 0110 \end{cases} \rightarrow \text{T}. \rightarrow \text{inc-iar} \\
T \rightarrow \text{Null}
\]

ADD AND SKIP (AS)

\[
\text{execute-as} = \text{inc-sdr mem-sdr create-ea}
\]

STORE REGISTER (STR)

\[
\text{execute-str} = \text{set-result acc-xr(j)}
\]

where

\[
\text{acc-xr(j)} = \text{s-xr(j)} : \text{s-word's-acc}
\]

LOAD REGISTER (LR)

\[
\text{execute-lr} = \text{set-result xr(j)-acc}
\]

where

\[
\text{xr(j)-acc} = \text{s-acc} : \text{s-word's-xr(j)}
\]

INTERCHANGE REGISTER (IR)

\[
\text{execute-ir} = \text{cnst-acc acc-xr(j) xr(j)-cnst}
\]
where
\[ \text{xr}(j) - \text{cnst} = s - \text{cnst} : s - \text{word}^o s - \text{xr}(j) \]
and
\[ \text{cnst} - \text{acc} = s - \text{acc} : s - \text{word}^o s - \text{cnst} \]
LEVEL EXIT (LEX)
\[
\text{execute-lex} = \\
\quad \text{s-run} : 0
\]
STOP (STP)
\[
\text{execute-stp} = \\
\quad \text{s-run} : 0
\]
CHAPTER BIBLIOGRAPHY

1. International Business Machines, IBM SYSTEM/7 Functional Characteristics, Form No. GA34-0003.


CHAPTER VI

HOW TO USE THE SYSTEM/7 SIMULATOR

This chapter discusses how to use the SYSTEM/7 Simulator (SIM/7). SIM/7 is written in PL/I and runs on an IBM 360/50 under OS-MFT. It executes in a 108 K partition. The standard IBM SYSTEM/7 Assembler (3), Linkage Editor (2), and Formatting (1) utilities are used to generate input for SIM/7. SIM/7 is executed in the batch mode and is run just as any other batch job. However, before executing SIM/7, the user must first have an understanding of how to use the simulator, what it offers, and how to actually set up the JCL for job execution. This chapter details how to use SIM/7.

Support Software

A simulator is only as useful as its support software will allow it to be. In this text support software is divided into two groups. The first consists of the software required to produce an executable object deck (i.e., assembler, linkage editor, and formatting utilities). The second consists of service functions useful in debugging during execution.

Since SIM/7 uses the standard IBM software for assembling, linking, and formatting, it is an extremely
useful simulator. By using the standard software, instead of writing special software for these functions, SIM/7 support software stays current. If instructions are added or deleted, the only software requiring modification is the simulator. If a special assembler, linkage editor, or formatting utility routines were involved, they too would require modification. A detailed description of the SYSTEM/7 Assembler, Linkage Editor, and Formatting utilities will not be given in this text. The IBM publications for these are listed in the bibliography (3). However, examples of their use could prove valuable. Figure 23 is an example of how to assemble, link, format, and execute a program using SIM/7.

The service functions are an important part of the support software for a simulator. They provide debugging facilities which are not found in the actual hardware. SIM/7 provides the following service functions:

1. Trace of Execution
2. Loader Dump
3. SNAP Dump
4. Input/Output Routines
5. Loader
6. Branch and Link Table

Each of the service functions is discussed at length in the following text. The Trace of Execution and Loader
Figure 23 - Job Control Language for Assemble, Link, Format, and Execute
dump are controlled by the Trace Control Card. The Trace Control Card is discussed at length in the text which follows.

**Trace Control Card**

The trace functions are initiated by the Trace Control Card (Figure 24). The double dollar signs ($$) beginning in column one are required. The Trace Control Card is input along with the object deck and must be the first card. The trace card parameters are explained in the following text along with the trace explanation. If no trace card is coded the full trace option is taken as default.

$$\text{Parm1, Parm2, \ldots, ParmN}$$

Figure 24 - Trace Parameter Card

There are four keywords which can appear as Trace Control Card parameters. The keywords are:

1. LVL = (Parm)
2. TIME = (Parm2, Parm2)
3. REGION = (Parm1, Parm2)
4. MEMORY = (Parm)

The keywords may appear in any order. Each keyword is followed by an equal sign and a parameter (5) enclosed in parentheses. If more than one keyword is used on
the trace card, the keywords and their associated parameters must be separated by commas as shown in Figure 25.

\[
\text{LVL}=(1), \text{REGION}=(0165,0208)
\]

Figure 25 - Trace Control Card Example

**LVL.** -- The LVL keyword specifies whether the Full or Partial trace (discussed later in this chapter) will be in effect. LVL=(1) specifies the Full trace and LVL=(2) specifies the Partial trace.

**TIME.** -- The TIME keyword specifies a span of time over which program execution will be traced. The TIME parameters are coded in micro-seconds and must be four digit decimal numbers.

**REGION.** -- The REGION keyword specifies a region of storage over which program execution will be traced. The REGION parameters are four digit decimal numbers.

**MEMORY.** -- The MEMORY keyword specifies that the binary input to the loader will be output. Currently the only valid MEMORY parameter is a one.

Trace of Execution

There are two options available for tracing the execution of a program. The first is the Full Trace which gives a detailed trace. The second is the Partial trace which gives limited information.
Full Trace

The Full Trace (LVL=1 on Trace Control Card) causes a detailed trace of each instruction to be output. The full trace can be turned on over the entire job (default), a specified span of time (TIME parameter), or a region of storage (REGION parameter). The full trace is the default if no LVL value is specified. Figure 26 shows a full trace of the sample program in the Appendix.
The image contains a table with hexadecimal values and some text above it. The table appears to be a part of a computer program or a similar technical document. The text above the table seems to be a note or instruction, possibly related to the table's content. The text is in English.
An explanation of the output of the full trace is contained in the following text.

**IAR.** -- Instruction Address Register. The IAR always points to the next instructions to be executed.

**I.** -- Hexadecimal representation of the instruction being executed.

**Operation.** -- The decoded mnemonic of the instruction being executed. If the instruction is a long instruction, the second field of the operation is the effective address generated for the instruction.

**XR1 through XR7.** -- Contents of index registers one through seven. The hexadecimal contents of each register are displayed after execution of the instruction.

**ACC.** -- Accumulator contents. The hexadecimal contents of the accumulator are displayed after execution of the instruction.

**RI.** -- Result Indicators. The results of the zero (Z), negative (N), positive (P), even (E), carry (C), and overflow (O) result indicators are displayed after instruction execution.

**Value Before and After.** -- If the contents of a location in storage are changed, the hexadecimal contents of the location 'before' and 'after' execution of the instruction are displayed.
State. -- The State of the machine is displayed after instruction execution. The processor states are described in Chapter III.

Time. -- Simulated execution time in micro-seconds.

Partial Trace

The partial trace (LVL=(2) on Trace Control Card) causes the IAR, Operation, and Time to be output. The IAR, Operation, and Time are defined with the full trace description. The partial trace can be turned on over the entire job (default), a specified span of time, or a region of storage. Figure 27 shows a Partial trace of the sample program in the Appendix.

<table>
<thead>
<tr>
<th>IAR</th>
<th>OPERATION</th>
<th>TIME</th>
<th>IAR</th>
<th>OPERATION</th>
<th>TIME</th>
<th>IAR</th>
<th>OPERATION</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0081</td>
<td>LI</td>
<td>8</td>
<td>0083</td>
<td>LXL</td>
<td>095</td>
<td>24</td>
<td>0085</td>
<td>LXL</td>
</tr>
<tr>
<td>00A5</td>
<td>BAL</td>
<td>48</td>
<td>00A6</td>
<td>STR</td>
<td>56</td>
<td>00A7</td>
<td>L</td>
<td>096</td>
</tr>
<tr>
<td>00A8</td>
<td>X</td>
<td>80</td>
<td>00AA</td>
<td>SZ</td>
<td>88</td>
<td>00AB</td>
<td>AI</td>
<td>100</td>
</tr>
<tr>
<td>00AC</td>
<td>AI</td>
<td>112</td>
<td>00AD</td>
<td>AI</td>
<td>124</td>
<td>00A6</td>
<td>B</td>
<td>00A6</td>
</tr>
<tr>
<td>00A7</td>
<td>L</td>
<td>152</td>
<td>00A8</td>
<td>X</td>
<td>09C</td>
<td>164</td>
<td>0080</td>
<td>B</td>
</tr>
<tr>
<td>0081</td>
<td>LI</td>
<td>188</td>
<td>0086</td>
<td>BR</td>
<td>086</td>
<td>196</td>
<td>0090</td>
<td>BN</td>
</tr>
<tr>
<td>0094</td>
<td>DMP</td>
<td>220</td>
<td>0095</td>
<td>LEX</td>
<td>228</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

END OF SYSTEM-7 SIMULATION RUN
DATE - 6/20/75
TIME - 10.25

Figure 27 - Partial Trace of Sample Program in the Appendix
Loader Input Trace

The loader input trace (Memory=(l) on Trace Control Card) causes the loader input to be output. The object deck is output in binary, five words per line. The memory address where the data is to be stored is printed to the left of each line. This trace function is primarily designed as a debugging tool for the programmer working on the simulator. Figure 28 shows a loader input trace for the sample program in the Appendix.

Branch and Link Trace

The Branch and Link Trace function is output automatically and is not under programmer control. The location of each branch and link instruction and the location to which the branch is made is output in hexadecimal notation. Figure 29 shows a branch and link table for the sample program in the Appendix.

Binary Dump of Output from Loader

```
0080 0110000000000100 1000100110000000 0000000010001011
0083 1000100100000000 0000000010001011 0101111000001111
0086 0100000000011000 0000000010001011 01000000010100
0089 0000000100100000 0100000000000000 00010000100000
  .  .  .  .  .  .
00AF 0011111100000000 0110000011111111 0011111100000000
```

Figure 28 - Loader Dump for Sample Program in the Appendix
Branch and Link Table

<table>
<thead>
<tr>
<th>FROM LOCATION</th>
<th>TO LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0086</td>
<td>00A5</td>
</tr>
</tbody>
</table>

Figure 29 - Branch and Link Table for Program in the Appendix

Input/Output Functions

The actual SYSTEM/7 assembler input/output instructions are not simulated in SIM/7 and therefore are considered illegal instructions. However, the programmer can simulate reading input data and printing output data through a pair of macro calls. The input macro is INP. The input macro uses the SYSTEM/7 assembler read instruction to input data. When it detects a read instruction the simulator branches to a subroutine which inputs the specified data into the proper locations and increments the IAR to bypass the data immediately following the macro. INP expands into the instructions listed below.

```
NAME  INP         STR, LEN
NAME  RDI         1,0,9,1
DC     A(&STR)    
DC     F(&LEN)    
MEND   
DC     data to be "input"
         (provided by user)
```

Figure 30 illustrates reading a string of characters. The label on the macro call is optional. The first parameter is the target location for the input and the second
parameter specifies the number of words to be read. The input data must follow the macro call and can be in any valid SYSTEM/7 assembler representation.

```asm
: Labl INP Data,6          Simulate reading data below
  DC 6C'sample input'    (provided by user)
: STP                     STOP
Data  DC 6F'0'           Input data is put here
```

Figure 30 - Input Macro

The output macro is DMP. DMP uses the SYSTEM/7 assembler write instruction to print data from memory. When the write instruction is detected by SIM/7 it branches to a subroutine which outputs the specified data to the line printer. DMP is expanded by the SYSTEM/7 assembler into the instructions listed below.

```asm
NAME DMP LOCS, LOCE
NAME RDI 0,0,9,1
DC A(&LOCS)
DC A(&LOCE)
MEND
```

The label on the macro call is optional. The first parameter is the beginning location of the data to be printed and the second parameter is the ending location of the data to be printed. The output format consists of the address in
hexadecimal notation of the beginning word on a line of output followed by up to twenty words of data in hexadecimal notation, per line. Figure 31 illustrates how to use the DMP macro.

```
Lab2  DMP  beg, beg+7
STP
```

Figure 31 - Output Macro

Error Messages

The error diagnostics on the actual SYSTEM/7 are minimal. The type of error which occurred is vague and the information output, pertaining to the error, is usually inadequate. The only information output is the location of the error, the Processor Status Word, the register contents at the time the error occurred, and the error code.

When an error occurs, it is helpful to get an English description of the error and a dump of the locations around the instruction in error. SIM/7 attempts to supply these.

When an error occurs during simulation, SIM/7 outputs an English description of the error; a hexadecimal dump of the fifty locations preceding the instruction in error and the fifty locations following it, a hexadecimal dump of
the contents of the ACC and the seven index registers, and the location of the instruction in error. This enables the programmer to evaluate the error more easily and intelligently.

Figure 32 is a comparison of an error which occurred on the SYSTEM/7 and an error which occurred during simulation.

**SYSTEM/7 ERROR OUTPUT**

ER03 0602 4000 F002
ER06 035F 0080 F065 02AF F002 1119 0003 036A 0612 0602 0612
ER06 036A 0080 1800 0000 0004 1119 0003 1517 1727 0610 04BF

**SIM/7 ERROR OUTPUT**

****JOB TERMINATED AT LOCATION 0086
****ILLEGAL OP CODE 00000****
PROGRAM TERMINATED DUE TO FATAL ERROR
DATE - 07/07/75
TIME - 15.48

ADDR HEXADECIMAL DUMP OF LOCATIONS 0054 TO 00B8
0054 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0055 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0056 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0057 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
005C 0000 0000 0000 0000 6004 8900 0095 8A00 009A 0000
0086 4018 008B 4028 0090 2000 0840 0901 009F 00A2 3004
0090 004C 0901 08A2 0CA5 3000 0096 F1F2 F3F4 F5F6 F7F8
009A 009B F1F2 F4F5 F6F7 F8F9 C508 E4C1 D340 D5C5 D8E4
00A4 C103 F006 C100 F200 4820 3806 7101 7201 74FF 4820
00AE 38F7 3F00 60FF 3F00 44F7 0000 0000 0000 0000 0000
00B8 0000

****HEXADECIMAL DUMP OF ACCUMULATOR AND INDEX REGISTERS
0004 0096 0098 0000 0000 0000 0000 0000

Figure 32 - Comparison of SYSTEM/7 Errors and SIM/7 Errors
CHAPTER BIBLIOGRAPHY

1. International Business Machines, IBM SYSTEM/7 Format/1, Form No. GC34-0004.

2. International Business Machines, IBM SYSTEM/7 Linkage Editors, Form No. GC34-0006.

3. International Business Machines, IBM SYSTEM/7 Macro Assemblers, Form No. GC34-0018.
CHAPTER VII

MAINTAINING THE SYSTEM/7 SIMULATOR

Chapter VII discusses aspects of the SYSTEM/7 Simulator necessary for program maintainence. Since the real SYSTEM/7 system is constantly being improved and refined, SIM/7 must be modified at times to keep pace with these changes. If SIM/7 were not kept current, its usefulness would be seriously diminished. Since the simulator is written in a higher level language (PL/I), most system modifications should be transparent and require no modification.

Modifying the Program

The primary requisite for modifying the simulator is a thorough understanding of the existing system. SIM/7 is a large system, and many of its parts are interdependent and interconnected, sometimes subtly. A seemingly minor modification that produces the correct result may also cause serious problems elsewhere. Sometimes side effects such as these are not easily detected and may not be noticed for a long period of time. One way to alleviate possible errors is to carefully check labels effected by modifications (a cross-reference map showing all references to each label is located at the end of the program listing).
Another important aspect of any modification is the careful documentation of any changes. SIM/7 is thoroughly documented. Any changes must be thoroughly documented and dated in order to maintain the credibility of the simulator. Incorrect comments are worse than no comments at all. Modifications may require changes in current documentation as well as addition to it. This should be performed carefully and completely.

Any modifications to the simulator should be noted in the historical section of the documentation as well as in the code. This gives the programmer a quick reference to what changes have been made and why.

Program Description

SIM/7 is written in PL/I (40 character set) for an IBM 360/50 running under OS. It executes in a 108 K partition. Overlays are not used. Comments at the beginning of the program include a Table of Contents which gives a brief description of each routine and instruction, and the page on which it is located. Tables containing instruction mnemonics, opcodes, execution times, and functional characteristics are also included. A historical section where modifications and corrections can be identified and dated for future reference is also included with the comments.
There are thirty-nine instructions (not including extended mnemonics) simulated in SIM/7. The opcodes are contained in an array labeled 'INSTOP'. The corresponding label for each instruction is contained in the label array 'CD'. Tables defining the variables in the program are included with the comments at the beginning of the program, and at the beginning of each subroutine.

Figure 33 is a block diagram of the SYSTEM/7 Simulator. The loader is described in Chapter IV. A detailed description of the Fetch Cycle, Effective Address Generation, and Instruction Execution is given in Chapter V. The Trace is described in Chapter VI.

Files

The SYSTEM/7 Simulator uses four files during execution. The four files constitute the input and output requirements for SIM/7. Each file is discussed in the following text.

DMP. - The DMP file contains the output from the Loader Dump, Snap Dump (output macro), and error messages. Any one or all of these output types may be on the file.

SYSPRINT. - The SYSPRINT file contains the output from the Trace service function.

SYSPUNCH. - Binary input to the SIM/7 loader. This is the storage module output from Format/7 (1).
Figure 33 - Flow Diagram of SIM/7
SYSIN. - The Trace Control Card(s) are input on the SYSIN file.

Functional Description of Instructions

A description of the SYSTEM/7 instruction set is given in Chapter III and a detailed breakdown of each instruction is shown. This section contains a table (Table II) which describes the functional characteristics of each instruction. This is presented here because it gives enormous insight into the internal activities of the simulator. If a modification is made, Table II will aid the programmer in determining which instructions are affected by the change. Questions are answered about each instruction as follows:

1. Result indicators affected?
2. Effected address generated?
3. Sign extension required (Immediate Instructions)?
4. Long instruction?
5. Contents of Accumulator and/or index registers are altered?
6. Contents of only the Accumulator are altered?
7. Memory contents changed?
<table>
<thead>
<tr>
<th></th>
<th>Changed</th>
<th>Memory is</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table II - Instruction Characteristics**
CHAPTER VIII

CONCLUSIONS

This thesis has examined general methods for simulating one computer system with another computer and has established guidelines which will be useful in other simulation projects. The methods and guidelines discussed were developed through research of other authors and through experience and knowledge gained by actually implementing a working simulator. A listing of the SYSTEM/7 Simulator is filed with the North Texas State University Computer Sciences Department. The SYSTEM/7 Simulator is the heart of this thesis and the remainder of this chapter evaluates the simulator and suggests future improvements for it.

Evaluation of the Simulator

The best way to evaluate the simulator is in terms of the original design goals established in Chapter II. A review of those goals shows that the simulator has fully succeeded in meeting them. The support software is powerful but simple to use. The assembler and loader software are standard IBM routines. The service functions described in Chapter VI are useful and easy to use. The assembler and
loader are standard IBM software and therefore require no maintenance. The simulator is modular in design and is written in a high-level language which makes modification and maintenance easier. The simulation is executed as a batch job and requires no special procedure for execution. This makes the user interface simple. The simulator is written in PL/I and runs in a 108 K partition which meets the memory requirements. It also executes at an efficient speed. The simulator has been used extensively, by the author, to test application routines which were used in a real-time application. The experience gained indicates that the system is accurate, foolproof, and easy to use and understand. The error messages are thorough, informative, and easy to read as shown in Chapter VI. Finally, the program listing is extensively commented; Chapters III, IV, VI, and VII of this thesis constitute a comprehensive users' manual for the system; Chapter V contains a detailed definition of the simulator using the Vienna Definition Language.

Areas for Future Development

The simulator has been used extensively by the author and many improvements have already been incorporated into the system. As the simulator is used by other programmers, further suggestions will be made. From these further improvements can be made. As the state-of-the-art advances,
new techniques may become evident, and these techniques may be used to enhance the effectiveness of the simulator.

Although the current version of the simulator performs excellently, the system will not be allowed to become stagnant. As new instructions and features for the SYSTEM/7 become available, they will be added to the simulator.

As the simulator system continues to evolve, new features will be added to its support software. One possible addition that may be desirable is the ability to collect statistics regarding the execution of SYSTEM/7 programs. For example, the frequency of execution of instructions or instruction types could be gathered. Using the simulator, just about any type of statistic could be gathered.

Another possible extension of SIM/7 would be the detection of infinite loops. Since the simulator is used at times by beginning programmers, infinite loops are not an uncommon occurrence. Therefore, it would be helpful to protect the system against such an occurrence.

Infinite loops can be created in various ways. One type of infinite loop involves an indirect addressing chain that eventually loops back onto itself thus creating a never-ending address calculation. Another type of infinite loop is when the program continually loops through a set of instructions and never exits. Yet another type of infinite loop can occur
when execution is transferred to an area containing data. In some instances data values may be interpreted as instructions causing not only infinite loops, but a variety of other problems.

One relatively simple method of loop detection is to count the number of instructions executed and compare the count to a predetermined value. When the count grows too large an error message can be printed and execution halted. It would be necessary for the user to be able to set the count in the event a large program was to be executed.

SIM/7 was written to execute in the batch mode. During development no terminal was available for use in designing and debugging terminal capabilities. The implementation of terminal capabilities in SIM/7 would be a valuable enhancement to the simulator.

With interactive capabilities the user could build a program file (either in batch or interactive mode) and then experiment interactively with the program for debugging. The user file could be identified by his or her social security number for easy identification. The simulator could then use the programmer's latest file without regard to whether the last run was made from batch or terminal. The addition of terminal (Timesharing) capabilities to SIM/7 would allow access to the simulator by remote users who are not equipped with card reader input into the system.
If terminal capabilities were added the full trace would have to be modified to print only eighty characters across the page instead of the one-hundred and twenty characters it prints on the line printer. The simulator would also have to be modified to search for a particular user file which would be used as input to the loader. This user file could be built either in the batch or interactive mode.

Another useful addition to SIM/7 would be the implementation of the input/output instructions. This would not be a trivial modification and would require much forethought before implementation. Many problems arise because of incompatibilities between the target and host peripheral devices.

A simulator rarely accomplishes an exact duplication of all of the target machines input/output instructions. Some input/output devices such as the teletype and disk can be simulated effectively. However, real-time input/output devices which create interrupts into the system at random can be difficult to simulate. For example, the SYSTEM/7 is designed to operate in real-time and accepts input from numerous peripheral devices as they interrupt the system. It would be impossible to exactly simulate all input/output devices in such a case. However, most input/output devices can be approximated by the simulator rather than duplicated.
The intended use of the simulator should dictate the level at which the input/output features are simulated. Since the SYSTEM/7 is designed to operate in real-time using an interrupt structure it would be of value to simulate the interrupts in some manner. One method of simulating an interrupting device is to build an array of times at which interrupts are to occur and use the times to trigger simulated interrupts. The array of times could be random times or times preset by the programmer. The simulator could test the simulated execution time against the interrupt times in the array and branch to a read routine (simulating an interrupt) when a match was found. In any event, it is evident that the simulation of the input/output instructions would be a challenging exercise.

No doubt many additional improvements, currently unforeseen, will be implemented as the system continues to evolve; however, this in no way implies that the current system is not adequate. It fully meets all of the design goals and has proved very useful. It has been thoroughly tested by the author, who has gained an enormous amount of knowledge through the work done on the system. However, as with any progressive system, the simulator must continue to be improved as the state-of-the-art advances. Future improvements therefore are inevitable.
APPENDIX

SAMPLE PROGRAM

The Appendix contains a sample program written in SYSTEM/7 assembler language. This program is used in explaining the various service functions in Chapter VI.

<table>
<thead>
<tr>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MACRO</td>
</tr>
<tr>
<td>2</td>
<td>&amp;NAME EQUAT</td>
</tr>
<tr>
<td>3</td>
<td>* EQUATE REGISTERS AND CONSTANTS</td>
</tr>
<tr>
<td>4</td>
<td>XR1 EQU 1 INDEX REGISTER 1</td>
</tr>
<tr>
<td>5</td>
<td>XR2 EQU 2 INDEX REGISTER 2</td>
</tr>
<tr>
<td>6</td>
<td>XR3 EQU 3 INDEX REGISTER 3</td>
</tr>
<tr>
<td>7</td>
<td>XR4 EQU 4 INDEX REGISTER 4</td>
</tr>
<tr>
<td>8</td>
<td>XR5 EQU 5 INDEX REGISTER 5</td>
</tr>
<tr>
<td>9</td>
<td>XR6 EQU 6 INDEX REGISTER 6</td>
</tr>
<tr>
<td>10</td>
<td>XR7 EQU 7 INDEX REGISTER 7</td>
</tr>
<tr>
<td>11</td>
<td>ACC EQU 0 ACCUMULATOR</td>
</tr>
<tr>
<td>12</td>
<td>MEND</td>
</tr>
<tr>
<td>13</td>
<td>MACRO</td>
</tr>
<tr>
<td>14</td>
<td>&amp;NAME INP &amp;STR,&amp;LEN</td>
</tr>
<tr>
<td>15</td>
<td>&amp;NAME RDI 1,0,9,1</td>
</tr>
<tr>
<td>16</td>
<td>DC A(&amp;STR)</td>
</tr>
<tr>
<td>17</td>
<td>DC F'&amp;LEN'</td>
</tr>
<tr>
<td>18</td>
<td>MEND</td>
</tr>
<tr>
<td>19</td>
<td>MACRO</td>
</tr>
<tr>
<td>20</td>
<td>&amp;NAME DMP &amp;LOCS,&amp;LOCE</td>
</tr>
<tr>
<td>21</td>
<td>&amp;NAME RDI 0,0,9,1</td>
</tr>
<tr>
<td>22</td>
<td>DC A(&amp;LOCS)</td>
</tr>
<tr>
<td>23</td>
<td>DC A(&amp;LOCE)</td>
</tr>
<tr>
<td>24</td>
<td>MEND</td>
</tr>
</tbody>
</table>
START 128

SAMPLE PROGRAM WHICH COMPARES TWO FIELDS

EQUATE REGISTERS AND CONSTANTS

EQU 1 INDEX REGISTER 1
EQU 2 INDEX REGISTER 2
EQU 3 INDEX REGISTER 3
EQU 4 INDEX REGISTER 4
EQU 5 INDEX REGISTER 5
EQU 6 INDEX REGISTER 6
EQU 7 INDEX REGISTER 7
EQU 0 ACCUMULATOR

LOAD WORD COUNT INTO ACC
LOAD ADDRESS OF FIELD1 INTO XR1
LOAD ADDRESS OF FIELD2 INTO XR2
BRANCH AND LINK TO SUB03
BRANCH TO EQUAL IF RESULT ZERO
BRANCH TO NEQUAL IF RESULT NEGATIVE
STOP - INVALID RETURN STATUS

OUTPUT USING DMP MACRO
GO TO END OF JOB
OUTPUT USING DMP MACRO
EXIT
STMT SOURCE STATEMENT

70 *
71 * ROUTINE WHICH COMPARSES TWO FIELDS
72 * AT ENTRY  ACC = NUMBER OF WORDS TO BE COMPARED
73 * XR1 = ADDRESS OF FIELD1
74 * XR2 = ADDRESS OF FIELD2
75 *
76 * ON RETURN  ACC = 0, FIELDS ARE EQUAL
77 * ACC = -1, FIELDS ARE NOT EQUAL
78 *
79 SUB03 EQU *
80 STR XR4 STORE WORD COUNT INTO XR4
81 S0302 EQU *
82 L 0,XR1 GET WORD FROM FIELD1
83 X 0,XR2 EXCLUSIVE OR WITH WORD IN FIELD2
84 SZ B S0309 SKIP IF ZERO
85 BR BRANCH - VALUES NOT EQUAL
86 S0304 EQU *
87 AI 1,XR1 INC XR1
88 AI 1,XR2 INC XR2
89 AI -1,XR4 DECREMENT XR4
90 SZ B S0302 SKIP IF ZERO
91 B S0302 LOOP BACK - CHECK NEXT WORD
92 S0306 EQU *
93 BR XR7 RETURN
94 S0309 EQU *
95 LI -1,ACC SET ACC TO -1
96 BR XR7 RETURN
97 END
BIBLIOGRAPHY


3. International Business Machines, IBM SYSTEM/7 Format/1, Form No. GC34-0004.


5. International Business Machines, IBM SYSTEM/7 Linkage Editors, Form No. GC34-0006.

6. International Business Machines, IBM SYSTEM/7 Macro Assemblers, Form No. GC34-0018.

7. International Business Machines, Linkage Editor and Loader, Form No. GC28-6538.


