Research and Development Technical Report
ECOM - 4529

COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE - FINAL REPORT, VOLUME IV - ARCHITECTURAL RESEARCH FACILITY; ISP DESCRIPTION, SIMULATIONS, DATA COLLECTION

Robert Gordon
Rosemary Howbrigg
Naval Underwater Systems Center

Susan Zuckerman
Naval Research Laboratory

Mario Barbacci
Daniel Siewiorek
Carnegie-Mellon University

September 1977

DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

ECOM
US ARMY ELECTRONICS COMMAND FORT MONMOUTH, NEW JERSEY 07703
NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government endorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.
Computer Family Architecture Selection Committee
Final Report
Volume 21, Architectural Research Facility; ISPL Description, Simulations, Data Collection.

Mario Barbacci, Daniel Siewiorek, Robert Gordon, Susan Zucker (Chair) Rosemary Morrow (Chair)
Center For Tactical Computer Sciences (CENTACS)
DRS-L-N-BC
Fort Monmouth, N.J. 07703

Approved for public release; distribution unlimited.

This report consists of a summary and nine (9) volumes. It is the result of joint Army/Navy work. The complete report may be separately published by the Naval Research Laboratories.


This volume describes the process of automatically gathering architectural data from the benchmark programs. Formal descriptions of the candidate architectures were written and run under a simulator. Assembly listings of the benchmark programs were used to generate simulation command files containing absolute code. These command files were then used to initialize the simulated memory. The result of the simulated benchmarks were collected into a data base for post-processing. Automating the data collection process not only eliminated tedious and potentially error prone hand calculations, but also provided the means to...
20. collect dynamic program behavior information that would be almost impossible to calculate manually.
# TABLE OF CONTENTS

## SECTION

<table>
<thead>
<tr>
<th>1. USES OF A FORMAL ISP DESCRIPTION</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2. WHAT IS ISP</td>
<td>2</td>
</tr>
<tr>
<td>a. Architecture vs. Machine Organization</td>
<td>2</td>
</tr>
<tr>
<td>b. Implementation Dependencies</td>
<td>3</td>
</tr>
<tr>
<td>3. SIMULATOR CHRONOLOGY AND CAPABILITIES</td>
<td>4</td>
</tr>
<tr>
<td>a. Chronology</td>
<td>4</td>
</tr>
<tr>
<td>b. Capabilities</td>
<td>4</td>
</tr>
<tr>
<td>4. THE CANDIDATE ISPS</td>
<td>6</td>
</tr>
<tr>
<td>a. ISP Seminars</td>
<td>6</td>
</tr>
<tr>
<td>b. Division of Labor</td>
<td>6</td>
</tr>
<tr>
<td>c. Correctness of the ISP Descriptions</td>
<td>6</td>
</tr>
<tr>
<td>5. DATA COLLECTION</td>
<td>8</td>
</tr>
<tr>
<td>a. Final Debugging</td>
<td>8</td>
</tr>
<tr>
<td>b. Preparation of Simulation Benchmarks</td>
<td>9</td>
</tr>
<tr>
<td>c. Counter Setting, Dumping, and Data reduction</td>
<td>10</td>
</tr>
<tr>
<td>d. Artificial Labels in the ISP Descriptions</td>
<td>10</td>
</tr>
<tr>
<td>6. THE CANDIDATE ISPS - READING HINTS</td>
<td>11</td>
</tr>
<tr>
<td>a. The ISP Description of the IBM S/360</td>
<td>11</td>
</tr>
<tr>
<td>b. The ISP Description of the Interdata 8/32</td>
<td>12</td>
</tr>
<tr>
<td>c. The ISP Description of the PDP-11</td>
<td>13</td>
</tr>
<tr>
<td>7. FUTURE USES OF ARF IN CFA</td>
<td>15</td>
</tr>
</tbody>
</table>

**APPENDIX A** - The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator

**APPENDIX B** - ISP Descriptions of the IBM S/360, Interdata 8/32, and DEC PDP-11

**APPENDIX C** - Sample ISP Simulator Session
1. USES OF A FORMAL ISP DESCRIPTION

Digital systems can be viewed as a hierarchy of levels: electronic circuit, logic-combinational and sequential, register transfer, programming, PMS (Processor, Memory, Switch), and network. For each of these levels there is a need to be formal, for communication purposes, and to have a representation or language that is convenient to use in the design process, so that concepts can be stated easily and analysis can take place.

ISP (Instruction Set Processor) was first introduced by [Bell and Newell, 1971] as a language for the programming level. Its initial goal was to describe computers in a systematic way and provide the reader with the information required to program the machine, excluding implementation details (e.g., memory speed, data path organization, etc.). Thus ISP can be used to describe a machine's architecture, as it is defined by the Computer Family Architecture (CFA) Selection Committee.

There are several uses of a formal architecture description in ISP. A sampling of some of these uses that are relevant to CFA are listed below:

a. Simulator. The architecture can be simulated/emulated and used to write and debug benchmark programs. Moreover, depending upon the simulator capabilities, it can be used to develop support and application software, and as a training device.

b. Architectural Evaluation. The ISP can be "instrumented" so that the relative efficiency of a candidate can be measured as a function of architectural parameters (e.g., the S, M, and R measures used by the Committee and described in Volume III). These architectural parameters are a function of the dynamic behavior of the benchmark programs. Such dynamic behavior is tedious and error prone to calculate by hand. Moreover, in some cases the dynamic behavior of a program may be impossible to model analytically and hence must be measured by instruction traces. Thus an instrumented simulator was the easiest, most accurate method of measuring dynamic program behavior.

c. Experimentation. Once the ISP is written, only moderate effort is required to make a perturbation to the description. Thus, effects of architectural changes can be debugged, measured, and studied without committing any funds to hardware development.

d. Procurement. Since the ISP is a concise definition of an architecture, it can be used as the basis of a procurement document for its implementation.

e. Verification. The ISP simulation of an architecture can be used as a standard to verify the correctness of a hardware implementation of the architecture. This is done by running verification programs on both the ISP simulation and the hardware implementation and comparing results.

Based on its advantages in the evaluation phase and its continued usefulness throughout the CFA project, ISP was selected to describe the three final candidates. Section 2 outlines ISP while Section 3 details the ISP simulator. Writing of the candidate ISPs and the benchmark program data collections are treated in Sections 4 and 5, respectively. Section 6 gives some further details and reading hints on the candidate ISPs. Finally, Section 7 depicts the future role of ISP in CFA. Appendix A is an ISP Compiler and Simulator User's Manual while Appendix B contains the ISPs for the three final candidates. Appendix C contains a sample session in which the simulator is used to execute one of the benchmarks.
2. WHAT IS ISP?

The ISP notation was developed to formalize the information normally given in basic machine manuals and if possible to supplement and eventually replace what are known as "programming reference manuals." Hence the essential requirements were of readability, completeness, flexibility, and brevity.

a. Architecture vs. Machine Organization

In a hierarchy of computer system descriptions there exists a level, the programming level, in which the basic components are the machine instructions, operations, and the interpretation cycle, all of which are defined in terms of lower level primitives, the so called Register Transfer Level.

The separation between a description and its lower level realization permits the design of "computer families" i.e. multiple realizations (mappings) of the same high level description in which the behavior of a processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules (a central processor). Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and the particular program. During the execution of a program, some set of bits (an instruction) is read from the main memory into an instruction register located in the central processor. This set of bits then determines the immediately following sequence of operations. After this sequence has occurred, the next instruction to be executed is determined and obtained, and the entire cycle repeats itself. This interpretation cycle is performed by a part of the processor called the interpreter.

Computers are usually described in ISP in terms of the following relatively fixed format:

(1) Memory - Physical components which hold information encoded in data. Among others, we have: Primary-Memory to hold programs and data, Processor-State and General Registers, Console-State to interface the processor with the operator, Input-Output-State to interface the processor with external devices.

(2) Data-Types - Which are described in terms of registers which could carry information.

(3) Data-Operations - Defining data transformations that can be carried out in terms of data-types.

(4) Instruction-Format - Specific instances of Data-types.

(5) Interpreter - The mechanism of the processor which fetches, decodes, and executes the instructions.

(6) Instruction-Set - The definition of the particular instructions that the processor executes.

This modularization of the description allows the designer to divide the processor in conceptually independent units - the actual hardware may or may not be implemented in that way.
b. Implementation Dependencies

ISP can be viewed as a programming language for certain class of algorithms i.e. Instruction Sets Processors (Architectures). Ideally, a language to describe architectures would not require the specification of any implementation details. Unfortunately, the ISP language does not aid the person writing the architecture description in distinguishing between truly architecture related items and implementation related items. The situation is similar to a common event in programming: A programmer describing an algorithm in a high level language is forced to take into account "implementation" details that are not part of the algorithm, e.g., the word length of the machine in which the program runs (it affects the result of the arithmetic operations). Due to the preciseness of ISP, it is necessary to define actions and registers which are not technically part of the architectural description. The mechanisms for doing functions, such as fetching instructions from memory, are not seen by the programmer. All she or he sees is the execution of an instruction or the trapping of error conditions such as exceeding memory boundaries. The methods for fetching the instruction, determining what it means, and starting the execution are left to the implementor. The only thing that the programmer is aware of is whether or not the instruction executes its given function and how it might fail. A complete and usable ISP description requires more than just the features a programmer would see.
3. SIMULATOR CHRONOLOGY AND CAPABILITIES

a. Chronology

During the spring of 1975 the Naval Research Laboratory contacted the Computer Science Department at Carnegie-Mellon University and expressed interest in the use of the ISP notation to write the formal specification of the CFA. CMU had been involved for several years in the development and use of ISP in design automation applications.

During the summer of 1975, NRL initiated the design of a system that would permit the "instrumented" simulation of the candidate architectures. The simulation facility was developed in several phases to minimize the risk of delays in the completion of the project. The compiler/simulation system was christened Architecture Research Facility (ARF) and is available on a nationwide basis through the computing facilities at NRL and CMU (the latter are available through the ARPA computer network, ARPAnet).

For the purposes of the selection process, the first three phases of ARF are of interest. ARF I was the enhancement of the ISP compiler available at CMU. This included expanding internal tables to allow the handling of large computer description, the implementation of better diagnostic facilities, and the addition of new features to the language. This phase was essentially completed by the end of 1975.

ARF II was a simulation facility based on an existing ISP simulator developed at CMU and was used to gather statistics from the benchmark programs. ARF II will be the facility described in the remainder of this section.

ARF III was a completely new simulation facility designed and implemented at NRL. It was completed and entered the testing phase while the selection committee was making its final decision. ARF III and its successors will continue to evolve and will constitute the basic tools needed for the modeling and verification of the chosen architecture in future CFA work.

b. Capabilities

A simulator is formed by linking the output of the ISP compiler with a table interpreting program. The simulator accepts commands from a teletype or user designated command file. The state of the simulator can be dumped to a command file which can be read at a future date when the simulation is continued. Command files can also be generated for initializing the target machine memory to contain a benchmark program (Section 5 describes how assembler output files were transformed into simulator command files).

The user interacts with the simulator through variable names and labels used in the ISP. The user can start and break the simulation on a label name. After a breakpoint the simulation is resumed on a continue command. Variables can have their values displayed or set under user control when the simulation is halted. During simulation, the successive assignments of values to traced variables are displayed on the user's terminal. Tracing allows the monitoring of the progress of a simulation and provides a very powerful debugging aid during the testing phase of an ISP.
For a complete description of the compiler and simulator capabilities the reader should consult Appendix A. A simple example of the use of the simulator is shown in Appendix C.
4. THE CANDIDATE ISPs

As part of the selection process, the Naval Research Laboratory funded a limited number of projects with the purpose of producing formal ISP descriptions of the candidate architectures. Carnegie-Mellon University has had many years of experience in computer description languages, particularly ISP, and provided expertise and tools for the task. The participation of CMU started during the spring of 1975 when initial plans were drawn for the implementation of a simulation facility that was to be used to verify the ISP descriptions and to execute selected benchmark programs. This simulated execution was to be fully instrumented and the measures obtained were used to compute the R and M measures of the candidate architectures.

a. ISP Seminars

Following the December CFA meeting (Fort Monmouth) it was decided to organize a seminar with the purpose of familiarizing the CFA committee members with the use of the notation. The seminar took place at CMU during the 14-16 of January 1976. Eighteen people participated, representing many of the member organizations. A second, smaller, seminar took place at CMU during the 25-26 of May 1976. This meeting was oriented specifically to discuss the ISP description of the three final candidates.

In addition to the formal seminars, several informal meetings took place at CMU and NRL. During these meetings plans for writing the ISP descriptions were considered and problems and programming techniques were studied.

b. Division of Labor

During the preliminary phases of the selection process it was decided that the task of producing an ISP description for a candidate architecture was the responsibility of the organization or subcommittee proposing the architecture. Early efforts from some members of the committee allowed some of the ISP descriptions to be initiated before the final candidates were selected. The Naval Underwater Systems Center (New London) was sponsoring the candidacy of the IBM S/36U-S/37U architecture and had already done some work on the description of a small 16-bit version of the IBM S/36U. Dr. Robert Gordon and Ms. Rosemary Howbrig of NUSC worked during the spring of 1976 on the full IBM S/36U description and by the time the final candidates were chosen the ISP description was close to completion. Dr. Daniel Siewiorek of CMU had completed a PDP-11 ISP description during the summer of 1975. This undertaking was part of CMU's work on computer description languages and applications. The PDP-11 description was subsequently modified and expanded according to the CFA requirements for statistic collections. The Interdata 8/32 presented a different picture. Ms. Susan Zuckerman of NRL started working on the ISP description of the 8/32 as late as April of 1976, but taking advantage of the accumulated experience on large ISP descriptions, particularly the IBM S/36U-S/37U, the Interdata 8/32 description was completed on time and the benchmarks for all three machines were processed before the 23 of July 1976 deadline.

c. Correctness of the ISP Descriptions

Perhaps one of the most serious questions to be asked is how correct are the ISP descriptions used in this project. The answer lies in the source of the information used to prepare the descriptions. All manufacturers provide
a "Principles of Operations" manual to aid the programmers. This manual is supposed to contain the true specification of the architecture. By far, the best documentation of the candidate architectures was provided by IBM, the least complete specification was that of the Interdata. Within IBM, the description of the architecture is the "Principles of Operation" manuals (i.e., an English description). DEC provides several manuals, one for each model, and this required going back and forth between manuals when details were not clear. The description of the Interdata required consultation with the manufacturer and some of the information was not guaranteed to be valid for later versions of the 8/32. We used the same documents, but being humans it is possible that we interpreted the English definition of the architecture differently from the implementors. All test cases which were run did agree with identical test cases run on real machines at Carnegie-Mellon University and Interdata. It must be remembered, though, that an ISP description is just another computer program and thus, if it is to be used, must be verified as being correct. This will require additional documentation of proprietary nature and architectural verification programs that the manufacturers have. This will have to be handled during the implementation phase, for the selected architecture.
5. DATA COLLECTION

Writing an ISP description for a large machine is not a trivial task. The candidate architectures had large instruction sets and although some features were excluded from the ISP, writing many hundreds of lines of code in a short period of time was a very satisfying and remarkable achievement when compared with software projects of similar magnitude.

In order to complete the task on time certain features of the candidate architectures were not described. These features were omitted on the basis of their importance to the CFA data collection phase. However, the ISP of the selected CFA will be fully specified and will contain all such features.

a. Memory Management - All three architectures have a virtual memory management mechanism, described in their principles of operations manual. By common agreement among the three architecture subcommittees this was considered a subsetable feature. The PDP-11 description already had this feature and was later used in one of the benchmarks. Neither the IBM S/360 nor the Interdata 8/32 descriptions have it.

b. Decimal Instructions - Only the S/360-S/370 offers this option. It was not needed to run the benchmarks. By common agreement among the architecture subcommittees it was deemed subsetable and therefore not included in the S/360-S/370 description.

c. Floating Point Instructions - All three architectures offer a Floating Point Instruction Set. Including this feature in the description would have greatly increased the time and manpower requirements for the task. The FP instructions are among the most complex instructions of any machine. By common agreement between the architecture subcommittees, floating point instructions were not included in the ISP descriptions. However, since some of the benchmarks required floating point operations, dummy procedures were included in the descriptions. This served a dual purpose, first it allowed us to keep the correct counts needed to compute the R and M measures; and second, it allowed the detection of those places where a benchmark executed a floating point operation and had to be helped around the trouble spot via simulation commands.

d. Error Handling - All three architectures define certain error recovery procedures (e.g., handling illegal operation codes, detecting address boundary errors, etc.). This feature was considered not crucial (the benchmarks were for the most part working programs that had already been executed on real machines) and it was up to the ISP writers to include it or not. The S/360-S/370 description contains a complete error handling mechanism, as defined in the principles of operations. The Interdata 8/32 description also has some error detection and recovery mechanisms. None were included in the PDP-11 description.

a. Final Debugging

All three descriptions were developed on the time-sharing facilities at CMU. The Advanced Research Projects Agency Computer Network (ARPANET) provided long distance access and the descriptions of the S/360-S/370 and the 8/32 architectures were written, debugged and tested directly from NRL and
NUSC (New London). For the final testing and running of the benchmarks, the people responsible for the descriptions met at CMU and the collection of statistics was performed in Pittsburgh.

Although most of the benchmarks were debugged and run on the real machines, other benchmarks were executed exclusively under the simulator. The latter included those programs using privileged instructions that were not directly available to non-system programmers (e.g., interrupt and I/O handlers). For the former set of benchmarks, results from the actual runs were available and used to check the simulated execution. For the second class of benchmarks the tracing and single stepping facilities of the ISP simulator were used to verify the correct execution of the programs. Breakpoints were used to detect the execution of non-implemented instructions (e.g., the Floating Point Set) and the simulated execution was guided around these instructions, taking care that the machine status and condition codes were properly set.

Although the ISP descriptions were essentially debugged before the benchmark execution phase started, there were some minor modifications and corrections that had to be done. These were performed concurrently with the data collection phase. The largest unforeseen problem was presented by the memory management feature of the PDP-11 which was based on the PDP-11/40 and had not been tested. One of the benchmarks (Quick Sort) called for a large address space and required the enabling of the feature. Unfortunately, the benchmarks had been tested on a PDP-11/45 which uses different unibus addresses for the memory management registers and this required minor modifications to the benchmarks. Most other problems were of a simpler nature and required only a few minutes to correct. It should be noted here that the simulator facility was also used to debug some benchmarks for the Interdata 8/32 before they were executed on the real machine. This was because no 8/32 was available near CMU and a large turn-around time (several days) would have complicated the debugging of the benchmarks.

b. Preparation of Simulation Benchmarks

The ISP simulator provides commands for the loading and initialization of the simulated machine memory and internal registers. The single most important feature of the command language which permitted the fast execution and collection of statistics was the ability to read "command" files containing the benchmarks to be executed. The command language can not handle programs in symbolic form (assembly language) and requires the pre-assembly of the programs into absolute, numeric, code. To this effect, a set of programs was developed at CMU which permitted the translation of assembly listing prepared by the real machine assembler into simulation command files. The operation was performed off-line.

Three sets of programs were prepared, one for each candidate architecture. The assembly listings were transported to CMU's PDP-10 using magnetic tapes (for the S/360 and the 8/32) or were prepared directly on the PDP-10 using a cross-assembler (for the PDP-11). The format of the assembly listings is different for all three machines. Nevertheless, in all three cases, it contains a listing of the relocatable object code. The procedure to translate this relocatable code into simulation command files consisted of the isolation of the code, the modification of the relocatable addresses using a user specified base address (multiple base addresses can be specified for the different control sections of the S/360), and the generation
of the ISP simulator commands loading the simulated machine memory locations with the code.

A total of 114 simulation runs were executed. They correspond to a total of 70 different benchmarks (some benchmarks called for several test cases, in other instances a benchmark had to be divided into separated sub-cases). The 70 benchmarks were divided as follows: 26 for the PDP-11, 22 for each of the IBM S/360-S/370 and Interdata 8/32. The appendix includes several examples of the command files used to simulate the benchmarks.

c. Counter Setting, Dumping, and Data Reduction

The ISP simulator permits the instrumentation of an ISP description by associating activity counters with each of the machine registers and memories. These counters allow the collection of statistics indicating the number of times each component of the machine is read from or written into. A normalized count is used and the counters are updated in terms of the number of 8-bit bytes actually involved in the operation. For registers with length different from a multiple of 8 bits the length count is rounded up (i.e., a 10 bit register operation counts as 2 bytes). A separate counter is kept for each label in the ISP description. Labels are included in the ISP descriptions to identify machine instructions, addressing modes, loops (used to describe vector-like instructions such as move character (MVC) on the S/360), as well as other ISP procedures. During the execution of the benchmarks, a database was created by collecting dumps of the counters after each benchmark was completed. The files containing the counters were then processed by other, off-line, programs in order to arrive at the H and R measures.

d. Artificial Labels in the ISP Descriptions

Certain modifications not normally needed were made to the ISP descriptions to aid in the collection of data during the running of the benchmark programs for the CFA project. Several labels and "do-nothing" procedures were added to allow easier measuring. These should not be looked at as necessary for the architecture description. A typical example of the need for the extra labels is given in the RX instructions of the S360: Register [0] can not be used as an index register and it was necessary to count the number of times that Register [0] was being specified as the Base or Index register. The labels added to count these events are clearly not part of the architecture or even the organization. Certain items, such as modifying the program counter during a branch operation or the setting of condition codes as a result of an instruction, were not to be measured in any of the three architectures for the CFA project. This required the addition of artificial labels that were used to identify portions of the description during which counting of events was disabled. This was typical of those actions which in a reasonable implementation would be done using ad-hoc circuitry, aside from the main operational units of the machine and thus, were not considered to affect the R measure.
6. THE CANDIDATE ISPs - READING HINTS

a. The ISP Description of the IBM S/360

In writing the ISP description of the IBM S/360 family of computers, a subset of the architecture was chosen. It included the entire standard instruction set, the protection feature instructions, and the direct-control feature instructions. It excluded floating-point and decimal instruction definitions. These choices were made due to limitations of time and personnel. It should be noted that IBM markets four instruction sets in the S/360 line. These are the Standard set, the Commercial set (Standard plus decimal), the Scientific set (Standard plus floating-point), and the Universal set (Standard plus decimal plus floating-point plus storage protection). Timer and direct-control features are additional options. Due to the upward compatibility between the IBM System/360 and the IBM System/370 lines, the generated description could be expanded to achieve the IBM System/370 description, but the level of effort required for this would be substantial.

For the purpose of extracting data for CFA decisions and for increasing the ease of running the simulator, several additional choices were made.

1. The diagnose instruction, which has a model dependent definition, was not written to directly correspond to any particular model. It was modified and used in aiding the termination of a simulation run.

2. Since several benchmark program authors wanted to use the compare logical long (CLL) instruction of the IBM System/370 architecture, it was added to allow for collecting data, but was not a true description of the instruction since it was not written as an interruptible instruction.

3. The test and set instruction was not described since no adequate mechanism in the ISP simulator allowed for a true execution of the mechanics of the instruction within one ISP program. A second parallel process should be defined for the main memory control unit. The same holds true for the I/O channel definition.

4. The front panel was minimal. Only a stop/run switch was included. Initial Program Loading (IPL), which is a front panel function, was not described.

Information necessary to write the description was obtained from the "IBM System/360 Principles of Operation" and the "IBM System/370 Principles of Operation" manuals. For the subset of the architecture described, it was not necessary to request further assistance and explanations from the manufacturer. Side effects of instructions were adequately described in the manuals. Model dependencies were also clearly enumerated. Instruction formats and addressing mechanisms were well defined and logically constructed. No ambiguities were discovered that couldn't be resolved using only the "Principles of Operation" manuals. This is not intended to imply that it would not be necessary to get further clarifications from IBM when describing the more complex supervisor state instructions in the System/370. In addition, we noted that some of the privileged state features of the System/370 are very model dependent and will probably be more so in the future. IBM may maintain compatibility at the problem state level only.
For reading ease, this ISP description consists of several sections:

Section 1: The first section contains declarations and is divided into two areas. First are the declarations which are part of the architecture description (i.e., those seen by a programmer). Next are implementation related variables which were items needed to adequately describe the architecture in ISP but are not seen by a programmer. It was divided in this way since the ISP language makes no distinction between truly architecture-related items and items necessary for a complete simulation of the architecture.

Section 2: The second section contains utility routines which were used throughout the description. Some routines are implementation related if they use implementation related variables.

Section 3: The third and largest section contains operand address generation routines and instruction descriptions. The instruction set is divided into four groups, each having a different amount of address generation required.

Section 4: The fourth section contains the interrupt processing description of the architecture. The order of handling the different classes of interrupts and the actual processing is thoroughly described in the IBM manuals. This is very unique in an architecture description from a manufacturer. A user gains a sense of reliability about the system, knowing that the real sequence of events that will occur on an interrupting condition allows the machine to recover from certain simultaneous hardware and software faults.

Section 5: The fifth section contains the instruction decoding and instruction cycle routines which fetch and execute an instruction. Everything to this point is considered to be declarations in the ISP language. Since all procedures must be defined before they are referenced, the last and smallest section contains only the executable program. The execution consists of doing one instruction and checking for interrupts whenever the CPU is not stopped; then repeating the cycle.

b. The ISP Description of the Interdata 8/32

The Interdata 8/32 ISP description is organized into seventeen sections. The description omits the I/O instructions, the floating point and double precision registers and instructions, and the MAC (memory access controller) operations. The ISP description is considered as both a simulation program (and therefore structured) and as a machine specification description.

Section 1: Interdata 8/32 storage resources as seen by the systems programmer (memory organization, register sets, program status word, instruction register).

Section 2: Temporary registers used for ISP description and implementation.

Section 3: ISP common subroutines used in later instruction descriptions.

Section 4: Interdata instruction format routines.
Section 5: Illegal instruction handler.
Section 6: Interdata LOAD and STORE instruction descriptions.
Section 7: BOOLEAN instructions.
Section 8: SHIFTS, TEST & SET, and TRANSLATE instructions.
Section 9: COMPARE, and CONVERT to HALFWORD VALUE instructions.
Section 10: BIT manipulation instructions.
Section 11: Arithmetic instructions.
Section 12: BRANCH instructions.
Section 13: CIRCULAR LIST instructions.
Section 14: Privileged Instructions and SUPERVISOR CALL instructions.
Section 15: Unimplemented instructions (floating point, I/O, double precision).
Section 16: Emulation routines for ISP (IFETCH, IXQT, INTCMK)
Section 17: Main instruction loop: EMULATE.

The Interdata 8/32 Manual omits discussion of instruction effects when non-standard (unexpected) parameters are specified in an instruction. For example: What happens if an odd register is given and the instruction expects an even register? What happens if memory boundary addressing is not adhered to? Conversations with Interdata personnel were needed to clarify these and other questions. The ISP description reflects the Interdata 8/32 operations as specified by the manual and personnel.

c. The ISP Description of the PDP-11

The PDP-11 line consists of 13 different models. In general the models are upward program compatible but there are instructions implemented in low end models that are not implemented in high end models and vice versa. The Initial PDP-11 description was modelled after the 11/40, a mid-range machine. Subsequently, the PDP-11/70 was specified by the CFA selection committee as the official PDP-11 architecture to be evaluated. Therefore the ISP was updated to incorporate the extra instructions.

Several features of the architecture were omitted from the description, and this situation will have to be corrected in the future: The floating point instructions, the interrupt mechanism, and the error detection and recovery mechanism. It should be noted here that there exist two different floating point instruction sets in the PDP-11 line, and that the memory management facility is not homogeneous across different models. These discrepancies will be resolved and an 11/70-compatible architecture will be specified in the final, formal ISP.
Following is a page by page description of the ISP:

Page 2-1. The primary memory and mappings (note word/byte memory and I/O page), central processor registers, and the floating point processor status register.

Page 3-1. The PDP-11/4U memory management registers and error registers that allow an instruction retry.

Page 4-1. Temporary registers not seen by programmer. These registers are necessary to completely define the algorithms performed by the hardware (such as address calculation) but these registers are not part of the architecture.

Page 5-1. Instruction decoding formats.

Page 6-1. Start of the procedures Memory accessing procedures.

Page 7-1. Effective address calculating procedures.

Page 8-1. Condition code setting procedures.

Pages 9 through 15. These are the actual instruction definitions. Similar instructions are grouped together into classes that follow the several levels of decoding that the hardware must go through.

Page 10-1. The instruction interpretation cycle.

During the course of the benchmark debugging and data gathering, several benchmarks made use of instructions not previously described in the ISP. These added instructions included SOB, MUL, DIV, ASH, and ASHC.
7. FUTURE USES OF ARF IN CFA

As stated in Section 1., Uses of a Formal ISP Description, ARF will continue to play a role in architectural experimentation, development of a procurement document, and verification of implementations of the CFA architecture. In addition, ARF and other ISP driven software tools are finding an expanding use in the DoD and ARPA community. Some of these uses include:

a. An emulator facility developed for the Air Force by the University of Illinois compiles code for a PDP-1U directly from an ISP description. An Intel 8080 runs at about a 300:1 simulation ratio. The facility will be used to debug and monitor large tactical programs for existing machines. It will also be possible to write code for machines before the hardware is available for users.

b. Even faster emulation speeds are possible when using microcode. TRW compiles microcode for a QM-1 from a SMITE description. They have achieved an 11:1 simulation ratio for an Intel 8080. SMITE is an ISP-like language and a program is being written to translate ISP descriptions into SMITE.

c. The potential for tools that operate relative to a computer description could represent a significant breakthrough in the manner that computer systems (hardware/software) are designed and evaluated. Currently effort is underway at Carnegie-Mellon University to develop a hardware design automation program and a compiler-compiler that take as input the symbolic description of a computer. Early results indicate that the resultant hardware design and generated code will be comparable to those produced by hand. Effort is also underway at Yale to automatically generate assemblers and I/O device handlers from computer descriptions.

d. Other areas in the early stages of development include automatic diagnostics generation, microcode generation, machine verification, and high level performance/reliability evaluators.

In the next five to ten years one can envision a system of programs that take as input computer descriptions and language and problem specifications, and from these, generate operating systems, compilers, and other support and application software automatically. Thus the entire proposed architecture could be evaluated without committing too many years of effort in both hardware and software design.

It is hoped that, with the software tools already developed, formal computer descriptions will play an increasing role in the Department of Defense's evaluation, procurement, verification, and programming of computers.
APPENDIX A

The Symbolic Manipulation of Computer Descriptions:
ISPL Compiler and Simulator

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh Pa.
August 2, 1976

This project is supported in part by the Advanced Research Projects Agency (ARPA) of
the Department of Defense, under contract F44620-73-C-0074, monitored by the Air
Force Office of Scientific Research, and by the National Science Foundation, under
grant GJ 32758X.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Declarations</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Memories and Registers</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Macros</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Identifiers and Constants</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Comments</td>
<td>9</td>
</tr>
<tr>
<td>3 Register Transfers</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Structure Selectors</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Transfers</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Shift Operators</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Arithmetic Expressions</td>
<td>14</td>
</tr>
<tr>
<td>3.5 Relational Expressions</td>
<td>15</td>
</tr>
<tr>
<td>4 Register Transfer Sequences</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Blocks</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Conditional Statements</td>
<td>17</td>
</tr>
<tr>
<td>4.3 Labelled Statements</td>
<td>19</td>
</tr>
<tr>
<td>4.4 The BAILOUT Operation</td>
<td>19</td>
</tr>
<tr>
<td>4.5 Statement-Lists</td>
<td>20</td>
</tr>
<tr>
<td>5 ISPL Programs</td>
<td>21</td>
</tr>
</tbody>
</table>

A-2

6 The Compiler Output ............................................ 24
   6.1 Running the Compiler ....................................... 24
   6.2 Example I - Listing ....................................... 25
   6.3 Example I - Symbol Table .................................. 25
   6.4 Example I - Cross Reference ............................... 28
   6.5 Example I - Statement Table ............................... 29

7 References ....................................................... 32

8 Appendix I - The Minicomputer Listing ......................... 33

9 Appendix II - ISPL Reserved Keywords ......................... 41

10 Appendix III - The XTOPI0.REQ File .......................... 42

11 Appendix IV - The Multiplier MACRO10 Format ............... 43

12 Appendix V - The SIMISP.REQ File ............................ 46
   12.1 The Statement Table ...................................... 46
   12.2 The Symbol Table ........................................ 47
   12.3 Table Diagrams ........................................... 48
A User's Guide to the ISPL Compiler

Mario R. Barbacci

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

ABSTRACT

The compiler described in this manual will translate programs written in a subset of ISP [Bell, 1971] into register transfer level instructions. The code thus generated could be used for the implementation of wiring list generators, simulators, or other Computer Aided Design applications. This manual describes the syntax and semantics of the language (ISPL) accepted by the compiler.

ACKNOWLEDGEMENTS

The compiler described here is an improved version of an original system implemented by S. Goldman and R. Scroggs. The syntax graph driving the compiler is generated using a program (GRPGEN) written by P. Karlton and R. Scroggs. This version of the manual reflects the modifications and improvements suggested by the users during the preparation of the ISP description of the candidate architectures for the Army/Navy CFA project. Special thanks are due to H. Elovitz (NRL), P. Gordon (NUSC), R. Howbrigg (NUSC), D. Siewiorek (CMU), and S. Zuckerman (NRL).
The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator

The Department of Computer Science at Carnegie-Mellon University is currently engaged in a research project exploring the uses of computer description languages in the automatic design of both software and hardware systems. This document describes a language, ISPL, based on the Instruction Set Processor notation of Bell & Newell [Bell, 1971]. The language was designed as a tool for the description of instruction sets i.e. the architecture of a computer, and has been used extensively in a design automation project at CMU [Siewiorek, 1976] and in the Army/Navy Computer Family Architecture Project.

Traditional computer description languages have been designed primarily for human communication and/or simulation. The SMCD [Barbacci, 1974] project has the more ambitious goal of developing design automation tools which would permit the generation of machine-relative software, documentation, hardware modular design, program verification, simulation, and generation of microcode. As in any evolutionary project, preliminary results are necessarily short of the ultimate goal; thus at this point we can present two concrete systems: a compiler and a simulator. A machine-relative compiler-compiler is being investigated by a group under W. Wulf. An automatic generator of hardware modular specifications is being developed by a group under D. Siewiorek and A. Parker. Further studies of computer descriptive languages are being carried out by this author and others.

As indicated above, the systems described in this report have been used as part of the Army/Navy CFA project, sponsored by the Army Electronics Command and the Naval Research Laboratory. Part of the project involved the description, in ISPL, of three commercial architectures: The DEC PDP-11, the IBM /360,370, and the Interdata
8/32. These descriptions were used to collect statistics on the execution of a set of benchmark programs under the ISPL simulator. Although the simulator is not particularly fast, its interactive facilities allow very exact control and detailed analysis of the register transfer operations being performed during the fetch/decode/execute cycle of the machines. The simulator was not meant to be used as a software development tool (although in fact, some CFA benchmarks for the Inte data 8/32 were debugged under the simulator, it being more accessible at CMU than the real machine), it is rather an Architectural Design tool that allows the user to explore alternative instruction sets and to collect statistics on the performance of the architectures.

Mario R. Barbacci
August 2, 1976


1. Introduction

The ISP (for Instruction Set Processor) notation was developed for a text [Bell, 1971] to precisely describe the programming level of a computer in terms of its memory, instruction format, data types, data operations, and a set of interpretation rules.

The behavior of a processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules (usually in the central processor). Thus if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and a particular program.

Although the above format is commonly used to describe a digital computer, ISPL is not intended to force the user into a given description style; ISPL can be used to describe register transfer systems in general (digital computers are a subset of such systems, namely those systems that interpret an instruction set).

The subset of ISP implemented by the compiler under discussion contains a number of features that allow the user to describe a wide variety of digital systems: Pseudo register declarations, macros, and compound statements. For efficiency reasons, certain other features described in [Bell, 1971] are not implemented. Among these are: multidimensional memory arrays, parameterized procedures, multiple word access, and scattered bit access. However byte access is implemented.

An ISPL program consists of a description of the memory components (memories and registers) and a description of the behavior of the system. Memory components are defined in ISPL by a name and a description of their structure using brackets to
group the subcomponents along a given dimension. In the current implementation the only subcomponents allowed are memory words and bits (as subcomponents of memory words and registers). The behavior of the system is given by a set of register transfer statements. These statements can be performed in sequence or concurrently.

In ISPL, concurrency of actions is the rule rather than the exception, and it is reflected in the use of ";" as a delimiter for lists of concurrent actions. Sequencing is expressed by using the term "next" as a delimiter for lists of sequential actions. Complex concurrent and sequential activities can be described in terms of simpler activities using "next", ";", "(" and ")" in an Algol-like block structure.

The ISPL compiler produces code for an idealized Register Transfer Machine. There are two types of instructions in the RTM: Data and Control instructions. Control instructions are used to sequence the operation of the machine. They contain instructions to START, STOP, BRANCH, DIVERGE into concurrent execution paths, etc. The Data instructions are used to define the Arithmetic and Logical operations among the registers of the machine. They are described in terms of a 3-address format:

\[ \text{destination} \leftarrow \text{source1 \ operation \ source2} \]

The RTM code produced by the compiler is presented in two formats. The first format is simply a tabular listing intended primarily for human use. The second format is intended primarily for machine consumption. The human intended tabular representation could be digested by suitable string manipulating programs and stored into a more convenient machine format. Several reasons argued against this approach: depending on the language used, writing these interface programs might involve a non trivial amount of work. Worse yet, any format modification intended to help human readers will render these programs obsolete. The solution adopted was to produce
another copy of the RTM code directly into a machine understandable format. Thus the
version of the RTM code intended for machine use is created as a "program" using
MACRO-10 as the intermediate language. The format of these programs is described in
the appendices.
2. Declarations

There are two types of declarations in ISPL: Memory Declarations (explained in this section) are used to describe the structure of the registers and memories in a machine; Procedure Declarations (explained in later sections) are used to describe the behavior of the functional units in a machine.

2.1. Memories and Registers

Memory components are defined in ISPL by a name and a description of their structure. The number of subcomponents at each level of decomposition is given by a bracketed list of constants, much like an array declaration in Algol.

The declarations are given by a list of individual component declaration using ";" as delimiter. There are two types of memory declarations: 1) A definition of a physical component (physical declaration), and 2) A definition of a logical component (logical declaration) in terms of a previously declared (physical or logical) component. A logical declaration uses the "::=" operator to make an equivalence between two components.
Examples

A<15:0>  Declares A as the name of a register 16 bits wide, named 15,...,0 (from left to right). The "::" or range operator is used to denote an abbreviated list of subcomponent names.

Mp[0:4095]<0:11>  Square brackets are used to specify those dimensions where the accessing is done through some "addressing" (switching) schema. The memory, Mp, consists of 4096 words, each of 12 bits, named (from left to right) 0,1,...,11.

R<15,13,11,9:10>  In general, the list of subcomponents along any dimension is given by a list of "names" for the individual subcomponents. Numbers used to name individual elements do not indicate relative position.

Mw[32767:0]<15:0>;  Now the designer can use either Mw (the "word" memory) or Mb (the "byte" memory).

Mb[65535:0]<7:0>;=Mw[32767:0]<15:0>;  The only concession to the use of numbers as both names and position indicators is by using the range ("::") operator, whereby the abbreviated list consists of the bounds and all integers in between, with the implication that these consecutive numbers also name consecutive (from left to right) elements. The use of an empty bit-list (<> indicates a single, unnamed bit.

Undeclared variables or multiple declarations of a variable are, usually, non-fatal errors. The compiler will warn the user if this situation arises. The compiler compares the lengths (Nwords*Nbits) of the left and right hand sides of a logical declaration; if the lengths do not match a warning is issued.

2.2. Macros

A different type of declaration, the MACRO declaration, allows the designer to
abbreviate the description by naming often used strings of characters. The macro name can then be used instead of the full string. The format of a macro declaration is the following:

MACRO identifier := any-string-of-characters-not-containing-a-$-sign $

Macros are handled in its entirety by the lexical phase, thus the parser never "sees" a macro expansion. Macros can, therefore, be declared at any point in the description, not necessarily in the declaration part, and remain in effect until the end of the description.

Examples

MACRO SIGNBIT := ACC<0> $

The use of SIGNBIT some time later in the description is equivalent to using ACC<0>. Macros are strictly in-line string substitutions.

A macro can be defined in terms of other macros and the user should be careful to avoid a recursive definition which would create a non-terminating string replacement loop.

There are implementation dependent limits on the size of a macro string. If a macro declaration exceeds this limit (1000 characters at present) a warning will be issued. Results might be unpredictable if this situation occurs.

2.3. Identifiers and Constants

An identifier in ISPL is a string of letter, digits, and "."s, beginning with a letter; the "." is included as an identifier character for readability purposes. In the current implementation only the first 6 characters of an identifier are kept by the compiler. Identifiers must, therefore, differ in the first 6 characters for the compiler to distinguish them. The lexical phase accepts upper and lower case ASCII characters but
they are converted and stored internally as upper case characters. This is another limitation of the implementation.

For readability purposes, identifiers can be followed by a larger and more descriptive version of the identifier. This secondary identifier is treated like an inline comment by the lexical phase. The syntax for this extended identifier use is:

```
short.identifier\this.is.a.long.identifier
```

An extended identifier can be appended to a short identifier using the "\" character. Such compound identifiers are valid wherever an identifier is valid. Notice that this is not the same thing as an "alias", as described in the full language [Bell, 1971]. The secondary name is stripped by the lexical phase and the designer must use the primary name for identification purposes.

Constants are strings of digits, interpreted as a number in some base. The default base is 10 (i.e., constants are decimal numbers unless otherwise specified). Constants in base 8 (octal numbers) must be tagged with the character #, as in #100 (decimal 64). Constants in base 2 (binary numbers) must be tagged with the character ', as in '100 (decimal 4). Constants in base 16 (hexadecimal numbers) must be tagged with the character '"', as in "A1 (decimal 161). The length of a constant is the minimum number of bits needed to represent it (i.e. leading 0's are stripped). The constant 0 is 1 bit long. The current implementation of the compiler limits constants to a maximum size of 35 bits.

2.4. Comments

Comments can be inserted in a description by preceding the comment string with the character "#". All characters following the "#" until the end of the line are ignored.
3. Register Transfers

Register Transfers are used to describe the data operations on the memories and registers (the data components) of the system. The syntax of a transfer follows very closely that of most programming languages. The main difference is the use of some special operators and the use of a non-standard operator precedence to accommodate these new operators.

The operators act upon the components of the system by taking the data stored in some components (the inputs), operating (i.e., transforming) on the data, and storing the resulting data in some component (the output).

The data used by the operators is defined in terms of the components that contain it. Since the memories and registers are declared as structured components made out of words and bits, a structure selector is needed in order to access or store data.

3.1. Structure Selectors

\[
\begin{align*}
\text{structure-selector} & ::= \quad \text{term} | \text{term} < \text{selector-range} > \\
\text{term} & ::= \quad \text{number} | \text{memory-access} | (\text{expression}) \\
\text{memory-access} & ::= \quad \text{identifier} | \\
& \quad \text{identifier} [\text{arithmetic-expression}] \\
& \quad \text{identifier} [\text{element-name}] \\
\text{element-name} & ::= \quad \text{number} \\
\text{selector-range} & ::= \quad \text{bit} | \text{bit} : \text{bit} \\
\text{bit} & ::= \quad \text{number}
\end{align*}
\]

The terms are the building blocks used in a register transfer expression. A term can be a constant, a memory-access (to select data stored in a memory or register), or an expression in parenthesis (thus allowing large and complex register transfer expressions).
A *structure-selector* is used to select parts of a term (i.e., to select bits of a register, a constant, or an expression). The nature of the register transfer operators requires that the operands be of homogeneous type (i.e., register-like) and length. Thus multiword memories must be accessed using an *arithmetic-expression* (the address calculation) enclosed in "[" and "]" to select one and only one word of the array.

The compiler compares the maximum value that the result of an address computation can have with the number of words declared for a memory. If the former exceeds the latter, a warning is issued.

When a *selector-range* is applied to a memory or register access term it must use the bit names used in the declaration. When it is applied to other types of term, whose structure has not been declared (i.e., constants and expressions), the bits of the term are implicitly named \( n, n-1, \ldots, 1, 0 \) (from left to right).

### Examples

- **ACC**
  - Select the entire ACC register
- **Mp[Pc]**
  - Select the word whose address is contained in register \( Pc \)
- **ACC<5>**
  - Select bit 5 of register ACC
- **Mp[R[INDEX]+DISPLACEMENT]<0>**
  - Select bit 0 of the word whose address is given by the effective address calculation expression
- **(A<7:0>+B<7:0>)<5:4>**
  - Select the 5th and 6th bits (from the right) of the result of the addition

Attempting to access undeclared bits of a register or memory word will result in a warning message. The compiler will then default the erroneous bit name to the leftmost bit of the declaration. When the selector range of a register or memory word attempts to switch the relative position of two bits, the compiler will switch the
selector range boundaries and issue a warning message. For instance, if X is declared as X<0:5>, both X<2:3> and X<3:2> are equivalent terms but in the second case a warning is issued.

3.2. Transfers

Register transfers are used to modify the contents of the registers and memories. The syntax of a transfer is the following:

\[
\text{transfer ::= } \text{memory-access } \rightarrow \text{arithmetic-expression} |
\]

\[
\text{memory-access } <\text{selector-range}> \rightarrow \text{arithmetic-expression}
\]

The use of a selector-range on the left hand side of the "\rightarrow" specifies a partial register (or memory word) modification; the non-selected bits are not disturbed. If the right hand side is shorter than the left hand side, the result is stored right justified and 0's are concatenated to its left to clear the high order bits of the left hand side. If the right hand side is larger than the left hand side truncation of the high order bits will occur (the compiler will issue a warning if this situation occurs).

The right hand side of a transfer is always an arithmetic-expression. The difference between an arithmetic-expression and an expression properly is in the use of relational operators, which are not allowed in the former. We will give more details in the subsection dealing with expressions.

3.3. Shift Operators

\[
\text{shift ::= } \text{structure-selector } |
\]

\[
\text{structure-selector } \rightarrow \text{shift-op structure-selector}
\]

\[
\text{shift-op ::= } \text{tSL } | \text{tSR } | \text{tSLO } | \text{tSRO } | \text{tSL1 } | \text{tSR1 } | \text{tRL } | \text{tRR } |
\]

\[
\text{concatenation ::= } @
\]

A-17
**ISPL Compiler: User’s Manual**

A *shift* is the first step in the hierarchy of register transfer operations, shift operators have the highest binding power (precedence). A *shift* always takes the following form:

```plaintext
left.operand shift-op right.operand
```

The meaning of the operators (all of them have the same precedence) is the following:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSL</td>
<td>Shift left the left.operand, one position, and insert the (rightmost bit of the) right.operand into the vacant position, dropping the leftmost bit of the left.operand. The length of the result is the same as the length of the left.operand. The result can be stored in a register or used as an operand when building complex expressions. The operator does not modify the left.operand, only the transfer operator (&quot;-&quot; ) can perform side effects.</td>
</tr>
<tr>
<td>ISSR</td>
<td>Shift right the left.operand, one position, and insert the (rightmost bit of the) right.operand into the vacant position, dropping the rightmost bit of the left.operand. The length of the result is the same as the length of the left.operand.</td>
</tr>
<tr>
<td>ISSLO</td>
<td>Shift left the left.operand the number of positions indicated by the value of the right.operand inserting 0’s in the vacant positions and dropping the rightmost bits of the left.operand. The right.operand is treated as an unsigned integer. The result has the same length as the left.operand.</td>
</tr>
<tr>
<td>ISSRO</td>
<td>Similar to ISSLO but shifting right.</td>
</tr>
<tr>
<td>ISSL1</td>
<td>Similar to ISSLO but inserting 1’s into the vacant positions.</td>
</tr>
<tr>
<td>ISSR1</td>
<td>Similar to ISSL1 but shifting right.</td>
</tr>
<tr>
<td>CRR</td>
<td>Rotate towards the right the left.operand by the number of positions indicated by the value of the right.operand. The length of the result is the same as the length of the left.operand.</td>
</tr>
<tr>
<td>CLR</td>
<td>Similar to CRR but rotating left.</td>
</tr>
<tr>
<td>@</td>
<td>Concatenate the left.operand with the right.operand. This operator is included among the shift operators for symmetry reasons. The length of the result is the sum of the lengths of the operands.</td>
</tr>
</tbody>
</table>
3.4. Arithmetic Expressions

- complement ::= shift | NOT shift
- conjunction ::= complement |
- disjunction ::= conjunction AND complement |
- negation ::= conjunction | disjunction OR conjunction |
- factor ::= conjunction EQV complement |
- sum ::= factor | sum MINUS factor |
- arithmetic-expression ::= sum

All logical operators (NOT, AND, EQV, OR, and XOR) operate on a bit by bit basis.

If the operands have unequal lengths the shortest operand is expanded (on the left) with 0's.

The arithmetic operators, with the exception of MINUS, operate on unsigned (pure magnitude) operands, the MINUS operator assumes a Two's Complement representation with a sign bit in the leftmost position. The main difference is in the padding used to match the length of their operands. The MINUS operator extends the sign of the shortest operand, the other operators use 0 as the padding character.

The length of the result of the infix operators "+", "-", and "MINUS" is one bit larger that the largest operand. The length of the result of the "*" operator is the sum of the lengths of the operands. The length of the result of the "/" operator is the same as the length of the left operand (the dividend).
3.5. Relational Expressions

In order to describe non-trivial systems, ISPL provides certain facilities to control the execution of the transfers. Thus certain transfers may or may not be executed depending on the result of some previous operation. These conditional activities are described in more detail in the following section. Here we are concerned with the basic data operators of the language, among which we include the relational operators used to build conditional expressions.

\[
\text{relation ::= } \begin{align*}
\text{arithmetic-expression} & | \\
\text{arithmetic-expression relop arithmetic-expression} & \\
\text{expression ::= } \begin{align*}
\text{relation} \\
\text{EQL} | \text{NEQ} | \text{LSS} | \text{LEQ} | \text{GEQ} | \text{GTR} | \text{TST}
\end{align*}
\]

Relational operators perform a test between their left and right operands. The result for all these operators, with the exception of TST, is a boolean value (TRUE or FALSE) which can be tested by one of the control operations defined in the following section. All relational operators treat the operands as unsigned integers. A 2's complement representation of a negative number will therefore look greater than a positive number of the same length.

The TST operator performs a logical subtraction of its operands and produces a result of 0, 1, or 2, indicating that the left operand is less than, equal to, or greater than the right operand, respectively.

Beware that relational operators have less precedence than logical and arithmetic operators, thus, the expression: A LSS B AND C GEQ D is parsed as: A LSS (B AND C) GEQ D which is syntactically incorrect. The proper way of writing the expression is: (A LSS B) AND (C GEQ D)

It was indicated before that the right hand side of a register transfer operation
(←) must be an arithmetic expression. This does not allow the use of relational operators. In order to use them on the right hand side of a transfer, the (relational) expression must be enclosed in parenthesis. This in effect transforms the (relational) expression into a term, a valid arithmetic expression, e.g.:

\[ \text{FLAG} = (A \neq B); \text{ Yields } 0 \text{ or } 1 \]
\[ \text{TVAL} = 1 + (D \text{ TST } E); \text{ Yields } 1, 2, \text{ or } 3 \]
4. Register Transfer Sequences

The behavior of a digital system is described in ISPL by a list of statements. These statements can be built up from register transfers by using two special delimiters to indicate sequential or concurrent execution. Statement lists can be nested using parenthesis to build more complex statement lists. The syntax of the register transfer sequences is as follows:

\[
\text{statement-list ::= parallel-statement-list | BAILOUT identifier | statement-list NEXT parallel-statement-list}
\]

\[
\text{parallel-statement-list ::= labelled-statement | parallel-statement-list ; labelled-statement}
\]

\[
\text{labelled-statement ::= statement | identifier := statement}
\]

\[
\text{statement ::= conditional-execute | conditional-decode | block | transfer | identifier}
\]

\[
\text{conditional-execute ::= ( IF expression => statement-list )}
\]

\[
\text{conditional-decode ::= ( DECODE expression => parallel-statement-list )}
\]

\[
\text{block ::= ( statement-list )}
\]

4.1. Blocks

Blocks are the simplest building tools to define complicated statements. A block is a statement-list enclosed in parenthesis:

\[
(A=0 \text{ NEXT } A A \text{ OR } B[X]<7:0> ; C+C+1)
\]

4.2. Conditional Statements

There are two ways of specifying conditional activities. These are the conditional-decode and the conditional-execute statements:

\[
( \text{ condition => statement(s) } ),
\]

A-22
where the conditions and their interpretation are as follows:

<table>
<thead>
<tr>
<th>CONDITION</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECODE <em>expression</em></td>
<td>The value of the expression is interpreted as an integer and used to select one out of n possible statements, given as a list of alternatives. These alternatives are separated by &quot;;&quot;, but in this case they are not considered to be concurrent activities; only one of them will be executed. The statements in the list are numbered 0 through n-1, from left to right. The ith statement is executed if the value of the expression is equal to i.</td>
</tr>
<tr>
<td>IF <em>expression</em></td>
<td>This is a special case of the conditional-decode statement. The statement-list following the ( \Rightarrow ) operator is initiated if the logical value of the expression is TRUE, otherwise it is bypassed. For simplicity, the expressions used in the conditional-execute statement do not have to be relational-expressions, yielding a TRUE or FALSE value. An arithmetic-expression can be used, with the implication that the result of the expression is tested against 0. The statement-list is executed if the expression is not equal to 0, it is bypassed otherwise. In other words, the expression is interpreted as ( \text{expression NEQ 0} ). For similar reasons, the conditional-decode statement accepts a relation as the conditional expression, with the implication that the logical values FALSE and TRUE are interpreted as the numbers 0 and 1, respectively.</td>
</tr>
<tr>
<td></td>
<td>The language does not provide an IF \ldots THEN \ldots ELSE type of conditional statement. They are trivially described using a 2-way DECODE statement. The user should be careful to write the alternative statements in the proper order: the 0th case (logical FALSE) first and the 1st case (logical TRUE) second. Thus the statements are reversed from the normal Algol-like order. Do not forget the &quot;;&quot;s after each alternative, except the last one, of a DECODE statement. A missing &quot;;&quot; in this context is a fatal error that is sometimes detected several lines after the offending alternative. The compiler will complain about a &quot;missing action list&quot;.</td>
</tr>
</tbody>
</table>

4.3. Labelled Statements

The statements described above can be identified with a label. This label is used to designate the starting point of the statement. The label of a statement can be used wherever a statement is valid. The interpretation given to the use of a label in the middle of a statement-list is the following:

1) If the label is associated with a procedure definition, it is interpreted as a call (invocation) of the procedure, unless the invocation occurs inside the definition of the procedure, in which case the invocation is interpreted as a jump to the starting point of the sequence (i.e. there are no recursive calls in ISP).

2) Other invocations are treated as jumps to the starting point of the sequence. In the current implementation, labels (and their sequences) need not be declared before they are used. Thus we can jump forward in the description.

A reserved label, STOP, is predeclared in the compiler. It can be used to indicate a jump to the end of the description.

4.4. The BAILOUT Operation

The BAILOUT operation provides a way to describe the handling of exceptional conditions that might occur during the fetching, decoding, and execution of instructions. This operation is in effect a super RETURN from a procedure when an exceptional condition arises. The BAILOUT operator is used together with the label of the procedure whose context we want to leave, i.e., BAILOUT returns accross multiple levels of (dynamically) nested procedures. For instance:

Examples

\[
p1 := (\ldots \text{NEXT} (\text{IF } x = y+z \text{ NEXT BAILOUT } p2) \text{ NEXT } \ldots)
\]

\[
p2 := (\ldots \text{NEXT } p1 \text{ NEXT } \ldots)
\]

\[
\text{Main} := (\ldots \text{NEXT } p2 \text{ NEXT } \ldots)
\]

A-23
In the above example, procedure MAIN invokes procedure P2 which starts execution of procedure P1. At some point, P1 decides that some error has occurred (IF X => ...) and that only MAIN can handle the situation. The effect of "BAILOUT P2" is to terminate the execution of P1 and P2 and return to procedure MAIN, at the point were it invoked P2.

4.5. Statement-Lists

Statements, labelled or otherwise, can be used to describe a list of concurrent activities, a parallel-statement-list, using the ";;" as delimiter. Parallel-statement-lists can be used to build sequences of activities or statement-lists, using the "next" operator as delimiter. Notice that the ";;" when used to indicate concurrency has a higher precedence than the "next" used to indicate sequentiality. For instance, in the following statement-list: A+B ;; C+D NEXT E+F the transfers A+B and C+D are executed concurrently, and only when they are both completed will the locus of control pass to the next statement, the transfer E+F.

One detail to keep in mind is that ISPL is a statement language, not an expression language (in the BLISS sense). In particular, there is no such thing as an empty or null sequence, thus sequences like: (A+B) or A+B; NEXT C+D are invalid (the ";;" must be followed by a statement). In some cases the compiler is capable of detecting the extra ";;" and will eliminate it after warning the user.
5. ISPL Programs

As mentioned in the Introduction, an ISPL description consists of a set of component declarations, together with a description of the behavior of the (main) system:

\[
\text{ispl-program ::= identifier ::= ( declaration-part statement-list )}
\]

The above syntax indicates that ISPL programs look like labelled blocks, with a declaration-part, local to the body of the block.

**EXAMPLE**

```
MULT:=
  (DECLARE
      MPD<15:0>; P<15:0>; C<15:0>;
      STEP := (DECIDE P<0> => P+P TSR 0; P=(P+MPD)<15:0> TSR 0
      ERALCED
    L0:=
      C=8 NEXT
      LI:=
        (STEP NEXT
           C-(C-1)<15:0> NEXT
           (IF C NEQ 0 => LI)
         )
    )
```

The first example presents the ISPL description of a simple 8-bit multiplier using the shift-and-add algorithm. The multiplicand resides in the leftmost 8 bits of the MPD register. The multiplier resides in the rightmost 8 bits of the P register. The partial product is developed using all 16 bits of the P register. Additional details about the algorithm can be found in [Bell, 1972].

The description begins with the specification of the label for the program (MULTIPLIER). Labels are used in ISPL to identify activities so that they can be branched to, or used as subroutines.
The program itself is enclosed in parenthesis, and consists of two parts. The declarations and the specification of the behavior. The former are specified as a list of individual component declarations (multiplicand, multiplier/product, and step counter), and one procedure (STEP) which performs the basic multiplication operation, using the reserved identifiers DECLARE and ERALCED as brackets. The specification of the activities of the system is given as a list of two sequential steps. The first step (C-8) initialises the counter and the second is given by a labelled (L1) block of activities. This consists of a sequence of three steps. The first one performs the basic multiplication operation by calling the procedure; the second step decrements the counter; the third step tests the counter to see if the operation has been completed. If the value of the counter has not reached 0 then a jump to the label is indicated by using the label (L1) as an activity. If the counter is 0 then control flows out of the labelled statement and reaches the end of the program.

The basic multiplication operation is described using the DECODE control operation. It implements a 2-way branch depending on the value of the expression P<0>. The alternative paths selected by this operation are given as a list using the ";" as delimiter. The first path (P-0) is selected if the value of the controlling expression (P<0>) is 0; the second path (P+(P+MPD) TSR 0) is selected if the value is 1. The operator TSR 0 represents a shift right inserting zero in the vacant position.
EXAMPLE

MINI := (DECLARE MEMORY AND REGISTERS
    M[0:A77] <11:0> ; \textit{MAIN MEMORY}
    Z<7:0> ; \textit{EFFECTIVE ADDRESS REGISTER}
    CACC<12:0> ; \textit{13 BIT ACCUMULATOR WITH CARRY POSITION}
    CARRY.BIT<1 : CACC<12>
    SIGN.BIT<1 : CACC<11>
    ACC<11:0> : CACC<11:0>
    IR<11:0> ; \textit{INSTRUCTION REGISTER}
    OP<11:0> := IR<11:9>
    I.BIT<0 : IR<8>:
    ADDRESS<7:0> := IR<7:0>
    IO.BITS<7:0> := IR<7:0>
    UCLASS<0 : IR<7>
    L<7:0> ; \textit{RETURN REGISTER}
    PC<7:0> ; \textit{PROGRAM COUNTER}
    IO.REG<7:0> ; \textit{INPUT-OUTPUT REGISTER}
    RUN<1> ; \textit{RUN MODE}
)
\smallskip
\begin{itemize}
\item \textbf{PROCEDURE TO INCREMENT PROGRAM COUNTER}
\item \textit{INCPC} := (PC<7:0>) \textit{NOTE THAT PC WILL WRAP}
\end{itemize}
\smallskip
\textbf{ERASED}

\begin{verbatim}
START := (DECODE RUN =>
  STOP;
  IF run==0
  (IF INCPC) NEXT INCPC NEXT
  (DECODE I.BIT = Z-ADDRESS ; Z=M[ADDRESS]<7:0>) NEXT
  (DECODE OP => \textit{INSTRUCTION DECODING}
    ACC-ACC AND M<2> ; \textit{AND}
    CACC-ACC + M<2> ; \textit{TAD (SETS CARRY BIT)}
    (M<2>=M<2>+1)<11:0> NEXT IF (M<2> EQL 0 => INCPC) ; \textit{ISZ}
    (M<2>+ACC NEXT ACC<0>) ; \textit{IDCA}
    (L=PC NEXT PC<2>) ; \textit{JSR}
    PC=Z;
    \textit{JUMP}
    IO.REG=IO.BITS;
    \textit{IOT}
  (DECODE UCLASS =>
    (IF IR<6 => INCPC) NEXT
    \textit{IF IR<5 => \texttt{ACC- NOT ACC) NEXT}
    \textit{IF IR<4 => ACC< ACC<0) NEXT}
    \textit{IF IR<3 => CACC-ACC+1) NEXT} \textit{SETS CARRY BIT}
    \textit{IF IR<2 => CACC-ACC-1) NEXT} \textit{SETS CARRY BIT IF BORROW}
    \textit{IF IR<1 => ACC< ACC< TS0 1) NEXT}
    \textit{IF IR<0 => ACC ACC< MS0 1) ; \textit{END OF UCLASS=0}
    \textit{IF IR<6 => INCPC) NEXT
    \textit{IF IR<5 => PC+1) NEXT
    \textit{IF IR<4 => PC-CACC<7:0>) NEXT}
    \textit{IF IR<3 => RUN=0) NEXT
    \textit{IF (IR<2 AND SIGN.BIT) OR}
    \textit{(IR<1 AND (ACC EQL 0)) OR}
    \textit{(IR<0 AND NOT SIGN.BIT)) => INCPC})
  )\textit{END OF UCLASS DECODING}
  \textit{END OF INSTRUCTION DECODING}
  \textit{END OF RUN=1 MODE}
  \textit{NEXT \textit{END OF INSTRUCTION CYCLE}
  )

\textbf{START}
\end{verbatim}

A-27
6. The Compiler Output

The compiler produces a listing file (with extension LST) and an "object code" file (with extension RTM). The latter extension stands for Register Transfer Machine. In other words, the compiler produces code for some idealized machine which executes register transfer operations.

6.1. Running the Compiler

The following example shows a typical execution. The actual calling procedure may change from installation to installation. When the compiler starts executing it prompts the user for the ISP source file name. If there are any error messages they are printed on the user's terminal as well as in the listing file. When the compilation is done (the compiler types messages indicating the current phase it is executing) it automatically calls the MACRO10 assembler and passes to it the name of the RTM file. At the end of the assembly the user should have the following files (assume the ISP source is called X.ISP): X.LST, X.RTM, X.REL, as well as the X.ISP file, of course.

```plaintext
ISP COMPILER Thursday 29 Jul 76 23:42:13 MULT.ISP (655MB25) PAGE 1
IP COMPILER Thursday 29 Jul 76 23:42:13 MULT.ISP (655MB25) PAGE 1
ru isp
Input File: mult.isp
Parse Completed.
Optimization Completed.
Semantic Check and Output Follows
ISP: NO ERRORS DETECTED
23:43:57
MACRO: MAIN
EXIT
```
6.2. Example I - Listing

The listing file reproduces the ISPL source program together with any warning and error messages. The listing file is organized in 4 parts: 1) The listing proper, 2) A cross-reference listing indicating the places in the RTM object code were the registers, memories, and labels are being used, 3) A symbol table listing containing all the user and system declared entities, together with their attributes, and 4) A statement table listing containing a readable version of the RTM object code.

6.3. Example I - Symbol Table

The compiler produced symbol table for the multiplier example is shown below. There is an entry (1 line) for each user or compiler declared component. These include memory components, labels, and constants. The INDEX column indicates the position in the symbol table of the entity. This index is used to represent the variables in the statement table.
The TYPE column describes the type of "variable" stored in a given entry of the symbol table. The valid types are: Memory Array (TYPE=1), Register (2), Constant (3), Label (4), Mask (5), Flag (6), Temporary register (7), and Temporary flag (10). The last two are used for compiler declared variables (for instance, temporary registers are declared in order to store partial results when evaluating expressions).

The FLAGS field contains information used by the compiler. It is displayed as part of the output mainly for debugging purposes (i.e. they show the status of the symbol table entry).

The DEF field is used to store a pointer to an associated symbol table entry. It is used when a memory component, say a register, is defined in terms of a previously declared memory component. For instance, we can declare:

```
INSTRUCTION.REGISTER<15:0>;
OP.CODE<3:0> := INSTRUCTION.REGISTER<15:12>;
```

In the symbol table listing, the DEF field for OP.CODE will point to a pseudo register declaration entry, corresponding to INSTRUCTION.REGISTER<15:12>. The DEF field for the latter will point to the main declaration of INSTRUCTION.REGISTER<15:0>.

If INSTRUCTION.REGISTER had been mapped on top of another register or memory

declaration, the DEF fields will chain these definitions. (DEF defines a chain of
definitions, the last entry of which is always the main declaration).

The LBL (LaBel) field associates with every user declared label, an integer used
by the compiler. This integer constitutes an internal label.

The BCNT and WCNT (Bit CouNT and Word CouNT, respectively) indicate the
number of bits and words for each memory and constant. (The count is given as an
octal number).

The PNAME (Print NAME) contains an identifier for each entry. For user
declared variables and labels it contains the identifier used in the program (truncated
to six characters). Constants are identified by their numeric value (octal). Masks are
represented as a pair of octal numbers. These indicate the left and rightmost bit
positions of the mask with respect to the right edge of the word (for instance, a binary
mask like 00011000 will appear as 4,3). System declared registers and flags are
given compiler generated names.

The last field of the symbol table, WORDS;BITS, contains the list of
subcomponents for each user declared memory or register. The list contains the bit
(word) names given in the declaration as well as the internal bit (word) names
generated and used for the compiler. The compiler generates a position dependent
internal bit (word) name which can be used to generate the proper subcomponent
accessing code. These position identifiers are indicated in parenthesis, next to the
user specified bit (or word) names.
ISPL Compiler: User’s Manual

6.4. Example 1 - Cross Reference

<table>
<thead>
<tr>
<th>INDEX</th>
<th>VAR</th>
<th>STATEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'C'</td>
<td>24 25 26</td>
</tr>
<tr>
<td>2</td>
<td>'L0'</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>'L1'</td>
<td>30 32</td>
</tr>
<tr>
<td>4</td>
<td>'MPD'</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>'MULT'</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>'P'</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>'STEP'</td>
<td>11 13</td>
</tr>
<tr>
<td>10</td>
<td>'STOP'</td>
<td>15 23</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>7 13 26</td>
</tr>
<tr>
<td>12</td>
<td>0, 0</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>17, 0</td>
<td>12 25</td>
</tr>
<tr>
<td>16</td>
<td>'ZFAAR'</td>
<td>26 27</td>
</tr>
<tr>
<td>17</td>
<td>'ZFAAR'</td>
<td>11</td>
</tr>
<tr>
<td>20</td>
<td>'ZFRAAB'</td>
<td>12 24 25</td>
</tr>
<tr>
<td>21</td>
<td>'ZFRAAC'</td>
<td>5 6</td>
</tr>
</tbody>
</table>
### 6.5. Example 1 - Statement Table

<table>
<thead>
<tr>
<th>INDEX</th>
<th>LABEL</th>
<th>FLAG</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SOURCE1</th>
<th>SOURCE2</th>
<th>MERGE</th>
<th>PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'MULT'</td>
<td>1</td>
<td>'START'</td>
<td>16</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>'MULT'</td>
<td>1</td>
<td>'SMERGE'</td>
<td>(5)</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>'ISP'</td>
<td>0</td>
<td>'ISP'</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>'STEP'</td>
<td>1</td>
<td>'SMERGE'</td>
<td>(7)</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>'ISP'</td>
<td>0</td>
<td>'RBYTE' 'STRAAC'</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(21)</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>'BRANCH'</td>
<td>0</td>
<td>'STRAAC'</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>'RSHIFT'</td>
<td>0</td>
<td>'P'</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>'JOIN'</td>
<td>0</td>
<td>'JOIN'</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>'ADD'</td>
<td>0</td>
<td>'ADD'</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(17)</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>'RBYTE'</td>
<td>0</td>
<td>'STRAAB'</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(20)</td>
<td>(17)</td>
<td>(15)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>'RSHIFT'</td>
<td>0</td>
<td>'P'</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>'STEP'</td>
<td>0</td>
<td>'STEP'</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>'RETURN'</td>
<td>0</td>
<td>'RETURN'</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>'L0'</td>
<td>1</td>
<td>'SMERGE'</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>'ISP'</td>
<td>0</td>
<td>'ISP'</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>'MOVE'</td>
<td>0</td>
<td>'MOVE'</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1)</td>
<td>(14)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>'L1'</td>
<td>0</td>
<td>'L1'</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>1</td>
<td>'SMERGE'</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>'ISP'</td>
<td>0</td>
<td>'ISP'</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>'CALL'</td>
<td>0</td>
<td>'CALL'</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>'DECR'</td>
<td>0</td>
<td>'DECR'</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>'RBYTE'</td>
<td>0</td>
<td>'RBYTE'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(17)</td>
<td>(1)</td>
<td>(15)</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>'NEQ'</td>
<td>0</td>
<td>'NEQ'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(17)</td>
<td>(1)</td>
<td>(11)</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>'IF'</td>
<td>0</td>
<td>'IF'</td>
<td>31</td>
<td>31,30</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>'JOIN'</td>
<td>4</td>
<td>'JOIN'</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>'SMERGE'</td>
<td>0</td>
<td>'SMERGE'</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>'NOOP'</td>
<td>0</td>
<td>'NOOP'</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>'NOOP'</td>
<td>0</td>
<td>'L0'</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>'NOOP'</td>
<td>0</td>
<td>'MULT'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>'STOP'</td>
<td>1</td>
<td>'STOP'</td>
<td>(10)</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The LABEL field is used to identify the individual statements.

The FLAGS field, as in the symbol table, is used internally by the compiler. In this particular example, the only flag shown indicates whether the label associated with the instruction was declared by the user (1) or by the compiler (0).

The OPR field contains the operation name. The meaning of most operations should be obvious from their names. Data operations are described as a 3-address assembly-like instruction. The source operands and the destination operand are indicated by their index into the symbol table (columns SRC1, SRC2, and DEST). The RBYTE operation is used to extract a byte from a register. The interpretation of the operation is the following: \( \text{DESTINATION} = \text{SOURCE1} < \text{SOURCE2} \) where destination and source1 are of type register and source2 is a mask. Other non-obvious data operations (not shown in the example) are:

- \( \text{WBYTE} (\text{DESTINATION} < \text{SOURCE1} > < \text{SOURCE2}) \),
- \( \text{READ} (\text{DESTINATION} = \text{SOURCE1}[\text{SOURCE2}]) \), and
- \( \text{WRITE} (\text{DESTINATION}[\text{SOURCE1}] < \text{SOURCE2}) \).

The RTM code uses at most three operands, thus an ISP statement like: \( A = B[C] < 1 > \) compiles into two RTM operations. The first is a READ operation that loads a (compiler generated) temporary register with \( B[C] \). The second operation is a RBYTE that extracts bit 1 of this temporary (the position of this bit is deduced from the declaration of \( B \)) and stores it into \( A \). Control operations are slightly more complex. Serial Merge (SMERGEOP) operations are used as merging points for non-concurrent sequences. Parallel merge (PMERGEOP) operations are used as merging points for concurrent sequences. Branch (BRANCHOP) operators select one out of many alternative sequences. These sequences are identified by a list of the labels of their
entry points, given in the same order as the conditional statement in the original ISP. Diverge (DIVERGEOP) operations are used to initiate simultaneous, concurrent paths. These paths are, as in the branch operations, indicated by a list of labels.

Branch and Diverge operations also specify the label of the statement following the alternative or concurrent paths. That statement is the "merge" point for the different paths.

The join (JOIN) operator is used as an unconditional jump statement. It generally appears as the last statement of a path, and jumps to the appropriate merging point (a serial or parallel merge). The NOOP operation is used as a control operation. It is generated by the compiler to indicate the end of a block. The statement points to the entry point of the block.
7. References


8. Appendix I ~ The Minicomputer Listing

```
[001] MINI: = (DECLARE MEMORY AND REGISTERS
[002] [M: (377): 11: 0; } \MAIN MEMORY
[002] Z<7:0>; \EFFECTIVE ADDRESS REGISTER
[002] CACC<12:0>; \13 BIT ACCUMULATOR WITH CARRY POSITION
[002] CARRY.BIT< > := CACC<12>
[002] SIGN.BIT< > := CACC<12>
[002] ACC<11:0> := CACC<11:0>
[002] IR<11:0>; \INSTRUCTION REGISTER
[002] 1.BIT< > := IR<8>
[002] ADDRESS<7:0> := IR<7:0>
[002] 10.BITS<7:0> := IR<7:0>
[002] UCLASS<7> := IR<7>
[002] L<7:0> \RETURN REGISTER
[002] PC<7:0>; \PROGRAM COUNTER
[002] IO.REG<7:8> \INPUT-OUTPUT REGISTER
[002] RUN<7> \RUN MODE
[002] ) \PROCEDURE TO INCREMENT PROGRAM COUNTER
[002] INCRC := (PC <- (PC+1]<7:0>) \NOTE THAT PC WILL WRAP
[003] EALCED
[003] START := (DECODE RUN =)
[004] STOP; \! If run=0
[004] (IR=M(PC) NEXT INCPC NEXT
[004] (DECODE 1.BIT = Z-ADDRESS ; Z=M(ADDRESS)<7:0>) NEXT
[004] (DECODE OP =) \INSTRUCTION DECODING
[004] ACC-ACC AND M(2); \AND
[004] (M(2)-M(PC)+1)<7:0> NEXT (IF 0=M<2> \INCPC) \ISZ
[004] (M(2)+ACC NEXT ACC<0>); \IDA
[004] (L-PC NEXT PC=2); \JSR
[004] PC<7> \JUMP
[004] IO.REG.IO.BITS; \!IOT
[004] (DECODE UCLASS =>
[004] ) \!END OF UCLASS DECODING
[004] ) \!END OF INSTRUCTION DECODING
[004] ) \!END OF INSTRUCTION CYCLE
[004] START
```

A-37
INDEX TYPE FLAGS DEF BLK LBL BCNT WCNT PWNAME WORDS:BITS:NAME(POSITION)
0 0 10000000 0 0 0 0 0 'e'
1 2 10100000 4 0 0 14 1 'ACC '<0(13):13(0)>
2 2 10010000 16 0 0 10 1 'ADDRESS '<8(7):7(0)>
3 2 10100000 5 0 0 1 1 'CACC '<13(0)>
4 2 10010000 5 0 0 10 1 'CACC '<8(13):13(0)>
5 2 10000000 8 0 0 15 1 'CACC '<8(14):14(0)>
6 2 10100000 5 0 0 1 1 'CACC '<14(0)>
7 2 10010000 6 0 0 1 1 'CARRY '.
10 2 10010000 15 0 0 1 1 'I.BIT '
11 4 10001100 0 0 3 0 0 'INCRPC'
12 2 10010000 20 0 0 10 1 'ID.BIT '<8(7):7(0)>
13 5 10000000 0 0 10 1 'ID.REG '<8(7):7(0)>
14 2 10100000 17 0 0 3 1 'IR '<12(13):13(0)>
15 2 10100000 17 0 0 1 1 'IR '<10(0)>
16 2 10100000 17 0 0 10 1 'IR '<8(7):7(0)>
17 2 10000000 0 0 0 14 1 'IR '<0(13):13(0)>
20 2 10100000 17 0 0 10 1 'IR '<8(7):7(0)>
21 2 10100000 17 0 0 1 1 'IR '<7(0)>
22 2 10000000 0 0 0 10 1 'L '<8(7):7(0)>
23 1 10000000 0 0 0 14 480 'M '<377(377):0(0):0(13):13(0)>
24 4 10000100 0 0 1 0 0 'MINI '
25 2 10010000 14 0 0 3 1 'OP '<12(13):13(0)>
26 2 10000000 0 0 0 10 1 'PC '<8(7):7(0)>
27 2 10000000 0 0 0 1 1 'RUM '
30 2 10010000 3 0 0 1 1 'SIGN.B'
31 4 10000100 0 0 10 0 0 'START '
32 4 10000101 0 0 10 0 0 'STOP '
33 2 10100000 21 0 0 1 1 'UCLASS '
34 2 10000000 0 0 0 10 1 'Z '<8(7):7(0)>
35 5 10000001 8 0 0 1 0 0 ', 0
36 3 10000001 0 0 0 1 0 0
37 3 10000001 0 0 0 1 0 0
38 5 10000001 0 0 0 1 0 0
39 5 10000001 0 0 0 1 0 0
40 5 10000001 0 0 0 1 0 0
42 5 10000001 0 0 0 1 0 0
43 5 10000001 0 0 0 1 0 0
44 5 10000001 0 0 0 1 0 0
45 5 10000001 0 0 0 1 0 0
46 5 10000001 0 0 0 1 0 0
47 5 10000001 0 0 0 1 0 0
48 5 10000001 0 0 0 1 0 0
50 10 10000001 0 0 0 1 0 0 'XTFRAA'
51 7 10000001 0 0 10 0 0 'XTFRAA'
52 7 10000001 0 0 10 0 0 'XTFRAA'
53 7 10000001 0 0 10 0 0 'XTFRAA'
54 7 10000001 0 0 10 0 0 'XTFRAA'
55 7 10000001 0 0 10 0 0 'XTFRAA'
56 7 10000001 0 0 10 0 0 'XTFRAA'
57 7 10000001 0 0 10 0 0 'XTFRAA'
60 7 10000001 0 0 10 0 0 'XTFRAA'
### INDEX

<table>
<thead>
<tr>
<th>INDEX</th>
<th>VAR</th>
<th>STATEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'ACC'</td>
<td>27 32 46 47 67 73 77 103</td>
</tr>
<tr>
<td>2</td>
<td>'ADDR'</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>'CARC'</td>
<td>32 77 103 130</td>
</tr>
<tr>
<td>7</td>
<td>'CARRY.'</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>'1.BIT'</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>'INCRPC'</td>
<td>7 18 43 63 120 151</td>
</tr>
<tr>
<td>12</td>
<td>'IO.BIT'</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>'IO.REG'</td>
<td>56</td>
</tr>
<tr>
<td>17</td>
<td>'IR'</td>
<td>15 61 65 71 75 101 105 111 116 122 126 132 136 141 145</td>
</tr>
<tr>
<td>22</td>
<td>'L'</td>
<td>51 124</td>
</tr>
<tr>
<td>23</td>
<td>'M'</td>
<td>15 22 26 28 31 34 37 40 40</td>
</tr>
<tr>
<td>24</td>
<td>'MINI'</td>
<td>160</td>
</tr>
<tr>
<td>25</td>
<td>'OP'</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>'PC'</td>
<td>5 6 15 51 52 54 124 130</td>
</tr>
<tr>
<td>27</td>
<td>'RUN'</td>
<td>12 134</td>
</tr>
<tr>
<td>30</td>
<td>'SIGN.B'</td>
<td>137 144</td>
</tr>
<tr>
<td>31</td>
<td>'START'</td>
<td>156 157</td>
</tr>
<tr>
<td>32</td>
<td>'STOP'</td>
<td>13</td>
</tr>
<tr>
<td>33</td>
<td>'UCLASS'</td>
<td>60</td>
</tr>
<tr>
<td>34</td>
<td>'Z'</td>
<td>20 23 26 31 34 37 40 40 52 54</td>
</tr>
<tr>
<td>35</td>
<td>'0,'</td>
<td>0 111 145</td>
</tr>
<tr>
<td>36</td>
<td>'0'</td>
<td>0 140</td>
</tr>
<tr>
<td>37</td>
<td>'1'</td>
<td>1 140</td>
</tr>
<tr>
<td>38</td>
<td>'1,'</td>
<td>107 111</td>
</tr>
<tr>
<td>39</td>
<td>'1'</td>
<td>113</td>
</tr>
</tbody>
</table>

A-39
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>105 141</td>
</tr>
<tr>
<td>41</td>
<td>2, 2</td>
</tr>
<tr>
<td>101</td>
<td>136</td>
</tr>
<tr>
<td>42</td>
<td>3, 3</td>
</tr>
<tr>
<td>75</td>
<td>132</td>
</tr>
<tr>
<td>43</td>
<td>4, 4</td>
</tr>
<tr>
<td>71</td>
<td>126</td>
</tr>
<tr>
<td>44</td>
<td>5, 5</td>
</tr>
<tr>
<td>65</td>
<td>122</td>
</tr>
<tr>
<td>45</td>
<td>6, 6</td>
</tr>
<tr>
<td>61</td>
<td>116</td>
</tr>
<tr>
<td>46</td>
<td>7, 0</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>130</td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>'/TRAAA'</td>
</tr>
<tr>
<td>41</td>
<td>4</td>
</tr>
<tr>
<td>140</td>
<td>142</td>
</tr>
<tr>
<td>51</td>
<td>'/TRAAN'</td>
</tr>
<tr>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>'/TRAAN'</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>53</td>
<td>'/TRAAN'</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>54</td>
<td>'/TRAAN'</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
</tr>
<tr>
<td>65</td>
<td>66</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
</tr>
<tr>
<td>75</td>
<td>76</td>
</tr>
<tr>
<td>101</td>
<td>102</td>
</tr>
<tr>
<td>105</td>
<td>106</td>
</tr>
<tr>
<td>109</td>
<td>111</td>
</tr>
<tr>
<td>112</td>
<td>116</td>
</tr>
<tr>
<td>117</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>126</td>
</tr>
<tr>
<td>127</td>
<td>132</td>
</tr>
<tr>
<td>133</td>
<td>136</td>
</tr>
<tr>
<td>137</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>142</td>
</tr>
<tr>
<td>144</td>
<td>146</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>60</td>
<td>'/TRAAN'</td>
</tr>
<tr>
<td>145</td>
<td>146</td>
</tr>
<tr>
<td>147</td>
<td></td>
</tr>
</tbody>
</table>

A-40

<table>
<thead>
<tr>
<th>INDEX</th>
<th>LABEL</th>
<th>FLAG</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SOURCE1</th>
<th>SOURCE2</th>
<th>MERGE</th>
<th>PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'START'</td>
<td>0</td>
<td>'SMERGE'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>'MINI'</td>
<td>1</td>
<td>( 24) 'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>'INCRC'</td>
<td>1</td>
<td>'SMERGE'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>'INCR '</td>
<td>0</td>
<td>INCR 'XTRAAC' 'PC'</td>
<td></td>
<td>( 57)</td>
<td>( 26)</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>'RBYTE '</td>
<td>0</td>
<td>RBYTE 'XTRAAC' 'PC'</td>
<td></td>
<td>( 26)</td>
<td>( 57)</td>
<td>( 46)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>'RETURN'</td>
<td>0</td>
<td>'INCRPC'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>'START'</td>
<td>0</td>
<td>'SMERGE'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>'ISP '</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A-41
37 0 'WRITE 'NM' 'Z 'XTRAAC'
    ( 23)( 34)( 53)
40 0 'READ 'XTRAAC''NM 'Z 'XTRAAC'
    ( 51)( 23)( 54)
41 0 'EQL ''XTRAAC''XTRAAC' 0
    ( 50)( 51)( 56)
42 0 'IF ' 'XTRAAC' 44 44,43
43 0 'CALL ' 'INCRPC' 3
    ( 50)
44 0 'SMERGE'
45 0 'JOIN'
46 0 'WRITE 'NM 'Z 'ACC'
    ( 23)( 34)( 1)
47 0 'CLEAR 'ACC'
    ( 1)
50 0 'JOIN'
51 0 'MOVE 'L 'PC'
    ( 22)( 26)
52 0 'MOVE 'PC 'Z'
    ( 28)( 34)
53 0 'JOIN'
54 0 'MOVE 'PC 'Z'
    ( 28)( 34)
55 0 'JOIN'
56 0 'MOVE 'IO.REG''IO.BIT'
    ( 13)( 12)
57 0 'JOIN'
58 0 'BRANCH' 'UCCLASS'
    ( 33)
61 0 'RBYTE 'XTRAAD''IR '6, 6
    ( 54)( 17)( 45)
62 0 'IF ' 'XTRAAD'
    ( 54)
63 0 'CALL ' 'INCRPC'
    ( 11)
64 0 'SMERGE'
65 0 'RBYTE 'XTRAAD''IR '5, 5
    ( 54)( 17)( 44)
66 0 'IF ' 'XTRAAD'
    ( 54)
67 0 'NOT ''ACC''ACC'
    ( 1)( 1)
70 0 'SMERGE'
71 0 'RBYTE 'XTRAAD''IR '4, 4
    ( 54)( 17)( 43)
72 0 'IF ' 'XTRAAD'
    ( 54)
73 0 'CLEAR 'ACC'
    ( 1)
74 0 'SMERGE'
75 0 'RBYTE 'XTRAAD''IR '3, 3
    ( 54)( 17)( 42)
<table>
<thead>
<tr>
<th>INDEX</th>
<th>LABEL</th>
<th>FLAG</th>
<th>OPCODE</th>
<th>DEST</th>
<th>SOURCE1</th>
<th>SOURCE2</th>
<th>MERGE</th>
<th>PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td></td>
<td>0</td>
<td>'IF'</td>
<td>'XTAAR'</td>
<td>100</td>
<td>100,77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td>0</td>
<td>'INCR'</td>
<td>'CACC'</td>
<td>'ACC'</td>
<td>(54)</td>
<td></td>
<td>(51)</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>2,2</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>0</td>
<td>'IF'</td>
<td>'XTAAR'</td>
<td>104</td>
<td>104,103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>0</td>
<td>'DECR'</td>
<td>'CACC'</td>
<td>'ACC'</td>
<td>(54)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>1,1</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>106</td>
<td></td>
<td>0</td>
<td>'IF'</td>
<td>'XTAAR'</td>
<td>110</td>
<td>110,107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>0</td>
<td>'RSHFT'</td>
<td>'ACC'</td>
<td>'ACC'</td>
<td>1</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>0,0</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>0</td>
<td>'IF'</td>
<td>'XTAAR'</td>
<td>114</td>
<td>114,113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>0</td>
<td>'LSHFT'</td>
<td>'ACC'</td>
<td>'ACC'</td>
<td>1</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>113</td>
<td></td>
<td>0</td>
<td>'SMF'</td>
<td>IR</td>
<td>'JOIN'</td>
<td>153</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>114</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>6,6</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>0</td>
<td>'IF'</td>
<td>'XTAAR'</td>
<td>121</td>
<td>121,120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
<td>0</td>
<td>'CALL'</td>
<td>'INCRPC'</td>
<td>3</td>
<td>(54)</td>
<td>(17)</td>
<td>(45)</td>
</tr>
<tr>
<td>121</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>5,5</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>122</td>
<td></td>
<td>0</td>
<td>'IF'</td>
<td>'XTAAR'</td>
<td>125</td>
<td>125,124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>4,4</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'PC'</td>
<td>'PCL'</td>
<td>(26)</td>
<td>(22)</td>
</tr>
<tr>
<td>125</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>3,3</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>126</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>2,2</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>131</td>
<td>131,130</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'PC'</td>
<td>'PC'</td>
<td>(26)</td>
<td>(51)</td>
</tr>
<tr>
<td>130</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'CACC'</td>
<td>'ACC'</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>131</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>3,3</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>132</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'XTAAR'</td>
<td>2,2</td>
<td>(54)</td>
<td>(17)</td>
</tr>
<tr>
<td>134</td>
<td></td>
<td>0</td>
<td>'SMA'</td>
<td>IR</td>
<td>'RUN'</td>
<td>'RUN'</td>
<td>(27)</td>
<td>(26)</td>
</tr>
</tbody>
</table>

A-43
<table>
<thead>
<tr>
<th>INDEX</th>
<th>LABEL</th>
<th>FLAG</th>
<th>_OPCODE</th>
<th>DEST</th>
<th>SOURCE1</th>
<th>SOURCE2</th>
<th>MERGE</th>
<th>PATHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>135</td>
<td>0</td>
<td>0</td>
<td>'SMERGE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>2, 2</td>
<td>( 54)</td>
<td>( 17)</td>
</tr>
<tr>
<td>136</td>
<td>0</td>
<td>0</td>
<td>'RBYTE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>'SIC'</td>
<td>( 54)</td>
<td>( 54)</td>
</tr>
<tr>
<td>137</td>
<td>0</td>
<td>0</td>
<td>'AND'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>'SIGN.B'</td>
<td>( 54)</td>
<td>( 54)</td>
</tr>
<tr>
<td>140</td>
<td>0</td>
<td>0</td>
<td>'EQL'</td>
<td>'XTRAAD'</td>
<td>'ACC'</td>
<td>0</td>
<td>( 50)</td>
<td>( 17)</td>
</tr>
<tr>
<td>141</td>
<td>0</td>
<td>0</td>
<td>'RBYTE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>1, 1</td>
<td>( 55)</td>
<td>( 17)</td>
</tr>
<tr>
<td>142</td>
<td>0</td>
<td>0</td>
<td>'AND'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>( 55)</td>
<td>( 55)</td>
</tr>
<tr>
<td>143</td>
<td>0</td>
<td>0</td>
<td>'OR'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>( 54)</td>
<td>( 54)</td>
</tr>
<tr>
<td>144</td>
<td>0</td>
<td>0</td>
<td>'NOT'</td>
<td>'XTRAAD'</td>
<td>'SIGN.B'</td>
<td>( 55)</td>
<td>( 30)</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>0</td>
<td>0</td>
<td>'RBYTE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>0, 0</td>
<td>( 50)</td>
<td>( 17)</td>
</tr>
<tr>
<td>146</td>
<td>0</td>
<td>0</td>
<td>'AND'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>( 50)</td>
<td>( 60)</td>
</tr>
<tr>
<td>147</td>
<td>0</td>
<td>0</td>
<td>'OR'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>'XTRAAD'</td>
<td>( 54)</td>
<td>( 54)</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>0</td>
<td>'IF'</td>
<td>'XTRAAD'</td>
<td>152, 151</td>
<td>( 54)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>0</td>
<td>0</td>
<td>'CALL'</td>
<td>'INCRPC'</td>
<td>3</td>
<td>( 11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>0</td>
<td>0</td>
<td>'SMERGE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>152, 151</td>
<td>( 54)</td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>0</td>
<td>0</td>
<td>'SMERGE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>152, 151</td>
<td>( 54)</td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>0</td>
<td>0</td>
<td>'SMERGE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>152, 151</td>
<td>( 54)</td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>0</td>
<td>0</td>
<td>'SMERGE'</td>
<td>'XTRAAD'</td>
<td>'IR '</td>
<td>152, 151</td>
<td>( 54)</td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>0</td>
<td>0</td>
<td>'NOOP'</td>
<td>'START'</td>
<td>10</td>
<td>( 31)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>0</td>
<td>0</td>
<td>'JOIN'</td>
<td>'START'</td>
<td>10</td>
<td>( 31)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>0</td>
<td>0</td>
<td>'NOOP'</td>
<td>'MINI'</td>
<td>1</td>
<td>( 24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>1</td>
<td>0</td>
<td>'STOP'</td>
<td>'STOP'</td>
<td>1</td>
<td>( 32)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9. Appendix II - ISPL Reserved Keywords

The following keywords and identifiers are reserved in the language:

AND
BAILOUT
DECLARE
DECODE
DELAY (not described in this manual)
EQL
EQV
ERALCED
GEQ
GTR
IF
LSS
LEQ
MACRO
MINUS
NEQ
NEXT
NOT
OR
STOP
TST
WAIT (not described in this manual)
XOR
10. Appendix III - The XT0P10.REQ File

XTTESTOP=#200,
XTEGLOP=#201,
XTNEG0P=#202,
XTLS00P=#203,
XTLE00P=#204,
XTGEO0P=#205,
XTG100P=#206,
XTNO0P=#210,
XTCLE00P=#211,
XTMO0P=#212,
XTM0Y0P=#213,
XTR0Y0P=#214,
XTRREAD0P=#220,
XTRWRIT0P=#221,
XTLR0TOP=#226,
XTRR0TOP=#227,
XTN0T0P=#238,
XTINC0P=#231,
XTDEC0P=#232,
XTLSHFT0P=#233,
XTRSHFT0P=#234,
XTAND0P=#235,
XTOR0P=#236,
XTXOR0P=#241,
XTER0P=#242,
XTADD0P=#243,
XTSUB0P=#244,
XTLSHFT10P=#245,
XTRSHFT10P=#246,
XTLSHFT80P=#247,
XTRSHFT80P=#250,
XTCONC0P=#251,
XTNE0G0P=#252,
XTSUBTH0P=#253,
XTM1ULT0P=#300,
XTD10V0P=#301,
XT1F0P=#300,
XTRETURNO0P=#351,
XTISPO0P=#352,
XTIPJ0IN0P=#353,
XTBAIL0UT0P=#361,
XTCALL0P=#363,
XTJOIN0P=#365,
XTBRANCH0P=#371,
XTDIVERGE0P=#372,
XTJOIN0P=#373,
XTMERGE0P=#374,
XTSTART0P=#376,
XTSTOP0P=#377,
Another version of the RTM code intended for machine consumption consists of a MACRO10 program in which all the information in the symbol and statement tables is encoded as MACRO10 statements (all of which are in fact, data definition statements).

In order to understand the RTM file (the ISP and listing files associated with this example were described previously, in the section describing the compiler output), the reader should have a working knowledge of BLISS10, enough to understand the SIMISP.REQ file describing the structure of the MACRO10 statements. The SIMISP.REQ file is given after the example.
The MACRO10 program starts by declaring certain symbols to be accessible to separately compiled modules. This is done with the INTERN MACRO10 operator. The symbols in question are the base address for the symbol and statement tables and the number of entries in each table (actually the index of the last entry, the first entry has index 0). The user therefore can access the symbol table entries between SYTABL[0,fieldname] and SYTABL[STTOP,fieldname] and the statement table entries between STTABL[0,fieldname] and STTABL[STTOP,fieldname].

The MACRO10 program is divided in two segments, the high segment contains the bit and word lists of the symbol table, as well as the label lists of the statement table. The low segment contains the symbol and statement tables properly.
The bit and word lists are declared as a list of expressions, using the EXP MACRO10 operation, each element of the list takes a full word on the PDP-10. Each bit and word list is identified by a label of the form: ?Bnnnn for bit lists and ?Wnnnn for word lists were nnnn is the index of the symbol table associated with the bit/word list. Every element of a bit/word list appears as a pair of consecutive elements in the EXP statement. The first (odd) element is the bit/word name. The second (even) element is the bit/word position. The bit/word list ends with a -1 as a bit/word name element.

The statement table label lists appear as lists of expressions, again using the EXP operation. These lists are identified by a label of the form $nnnnn were nnnn is the index of the statement table associated with the label list. There is no need for a special list terminator, the statement table entry contains a count or vector length for its label list, if any.
12. Appendix V - The SIMISP.ROD File

12.1. The Statement Table

MACRO

STFLAGS=0,27,9$, "ASSORTED FLAGS FOR THE STATEMENT
STOPERATION=0,18,9$, "OPERATION CODE. SEE XTOP.REQ
STOPAR=0,0,18$, "PARALLEL OPERATION CODE
STDESTINATION=1,24,12$, "DESTINATION VARIABLE SYMBOL TABLE INDEX
STSOURCE=1,12,12$, "SOURCE1 VARIABLE SYMBOL TABLE INDEX
STSOURCE2=1,0,12$, "SOURCE2 " " " "
STSCOUNT=2,18,18$, "NUMBER OF ELEMENTS IN STSLIST.
STLABEL=3,0,18$, "SYMBOL TABLE INDEX OR 0.
STMERGETABLE=2,0,18$, "LABEL OF THE ASSOC. MERGE STATEMENT FOR NEXTDIVERGE,XTBRANCH AND XCTCALL OPS.
STMERGETABLE2=1,2,12$, "LABEL OF ASSOC. STATEMENT FOR XCTALLOP.
STSLIST=3,18,18$, "POINTER TO VECTOR OF SUCCESSOR STATEMENTS.
STGROUPSTSTRUCT IS MAPPED ONTO THE VECTOR

BIND

!THE STTABLE FLAGS
STUSERLAB=118, "STATEMENT LABEL WAS DECLARED BY USER
STBREAK=111, "BREAK FLAG. SIMULATOR BREAKS AFTER FLAGGED STATEMENTS ARE EXECUTED
STTRACE=112, "TRACE FLAG. SIMULATOR WILL PRINT VARIABLES AFTER EXECUTION.
STRECORD=113, "RECORD THE SIMULATED TIME OF EACH EXECUTION
STIGNORED=114, "FLAGS DIVERGE,MERGE AND ASSOC. JOINS AS DELETED STATEMENTS!!
STOPAUSE=115, "DISABLES READ/WRITE/ACCESS TALLY
STECCETC=0; "ADD ANY OTHER FLAGS YOU LIKE

BIND

STENTRYSIZE=4; "14 WORDS/ENTRY

STRUCTURE STSTRUCTURE(INDEX,WORD,P,S)=(.STSTRUCTURE+.INDEX=STENTRYSIZE+WORD)<.P,.S>

EXTERNAL STSTRUCTURE STTABE; "THE STATEMENT TABLE
EXTERNAL STTOP; "THE INDEX OF THE LAST STABLE ENTRY (STARTING FROM 0)

MACRO

STSUCLABEL=18,18$, "THE SUCCESSOR LABEL
STSUINDEX=0,18$, "THE SUCCESSOR INDEX

STRUCTURE STSUSTRUCTURE(WORD,P,S)=(.STSUSTRUCTURE+WORD)<.P,.S>
12.2. The Symbol Table

MACRO

\begin{verbatim}
SYSTYPE=0,27,9$, "THE ENTRY TYPE (1=MEMORY, 2=REGISTER, 3=CONSTANT, 4=LABEL, 5=MASK, 6=FLAG, 7=REGISTER, 10=TFLAG)"
SYFLAGS=-1,18,9$, "ASSORTED FLAGS FOR THE ENTRY"
SYDEFINITION=8,0,18$, "INDEX OF ASSOCIATED ENTRY. USED FOR REG-DEFINITIONS"
SYLABEL=1,18,18$, "INTERNAL STATEMENT TABLE INDEX FOR ENTRIES OF TYPE 4"
SYBITCNT=1,0,18$, "NUMBER OF BITS/WORD OR CONSTANT LENGTH"
SYWOPT2=2,18,18$, "POINTER TO WORD LIST (ONLY FOR TYPE 1)"
SYBIFPTT=2,0,18$, "POINTER TO BIT LIST (ONLY FOR TYPE 1 OR 2)"
SYBIFNAME=3,0,36$, "A SIXBIT STRING FOR VARIABLES, VALUE FOR CONSTANTS AND MASKS (LEFTBIT, RIGHTBIT)"
SYWRCN=4,0,36$, "NUMBER OF WORDS (ONLY FOR TYPE 1)"
\end{verbatim}

BIND

\begin{verbatim}
SYENTRIESIZE=5, "WORDS/ENTRY"
SYSYSTEMVAR=110, "SYSTEM DECLAIMED VAR. (TYPE 3, 5, 7, 10)"
SYBREAK=111, "BREAK FLAG. USED ONLY FOR LABELS"
SYTRACE=112, "TRACE FLAG. SIMULATOR TELLS AFTER VARIABLE IS WRITTEN INTO"
SYPRIMAY=114, "INDICATES VAR. IS LEFT HALF OF REG-DEFINITION"
SYSECONDARY=115, "INDICATES VAR. IS RIGHT HALF OF REG-DEFINITION"
SYBITADDRESS=116, "INDICATES STORAGE IS BIT ADDRESABLE"
TYPMEMORY=1, "FOR SYTYPE ABOVE"
TYPeregister=2, "" ""
TYPERECONSTANT=3, "" ""
TYPETELABEL=4, "" ""
TYPETEMP=5, "" ""
TYPETFLAG=6, "" ""
TYPETREGISTER=7, "" ""
TYPETFLAG=8, "" ""
\end{verbatim}

\begin{verbatim}
STRUCTURE SYSTRUCTURE(INDEX,WORD,P,S)=L(STRUCTURE+.INDEX+SYENTRIESIZE+.WORD)<P,S>)
\end{verbatim}

EXTERNAL SYSTRUCTURE SYTABLE; "THE SYMBEL TABLE"
EXTERNAL SYTOP; "THE NUMBER OF ENTRIES - 1 (I.E. MAX INDEX)"

\begin{verbatim}
STRUCTURE ISVECTOR(NDX)=11 (..IVECTDR+.NDX)<8,36>;
\end{verbatim}

EXTERNAL ISPIT,ISPFPNAME,ISPEXT,ISPPPN,ISPDAT,ISPTIM,ISPVER;
### 12.3. Table Diagram

<table>
<thead>
<tr>
<th>SYTYPE</th>
<th>SYFLAGS</th>
<th>SYDEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYLABEL</td>
<td>SYBITCNT</td>
<td></td>
</tr>
<tr>
<td>SYWDPTR</td>
<td>SYBITPTR</td>
<td></td>
</tr>
<tr>
<td>SYWDNAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYWDCNT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| FIRST WORD/BIT NAME | |
| FIRST WORD/BIT POSITION | |
| | |
| LAST WORD/BIT NAME | |
| LAST WORD/BIT POSITION | |
| | -1 |
A User's Guide to the ISPL Simulator

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>From ISPL to RTM and Beyond</td>
</tr>
<tr>
<td>3</td>
<td>The Command Language</td>
</tr>
<tr>
<td>3.1</td>
<td>START and CONTINUE</td>
</tr>
<tr>
<td>3.2</td>
<td>EXIT</td>
</tr>
<tr>
<td>3.3</td>
<td>READ and DUMP</td>
</tr>
<tr>
<td>3.4</td>
<td>ECHO and DECHO</td>
</tr>
<tr>
<td>3.5</td>
<td>RADIX</td>
</tr>
<tr>
<td>3.6</td>
<td>CTR, SETCTR, and OUTCTR</td>
</tr>
<tr>
<td>3.7</td>
<td>OPAQUE and DOPAQUE</td>
</tr>
<tr>
<td>3.8</td>
<td>VALUE and SETVALUE</td>
</tr>
<tr>
<td>3.9</td>
<td>TRACE, UTRACE, DTRACE, and TELLTRace</td>
</tr>
<tr>
<td>3.10</td>
<td>BREAK, DBREAK, and TELLBREAK</td>
</tr>
<tr>
<td>3.11</td>
<td>SBREAK, DSBREAK, and TELLSBREAK</td>
</tr>
<tr>
<td>3.12</td>
<td>ICONNEct and OCONNEct</td>
</tr>
<tr>
<td>3.13</td>
<td>HELP</td>
</tr>
<tr>
<td>4</td>
<td>Storage Mapping</td>
</tr>
<tr>
<td>4.1</td>
<td>Allowable Types of Mapping</td>
</tr>
<tr>
<td>5</td>
<td>Examples</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>5.1</td>
<td>Linking the Compiler Output with the Simulator</td>
</tr>
<tr>
<td>5.2</td>
<td>Running the Simulator</td>
</tr>
<tr>
<td>5.3</td>
<td>Executing Selected Procedures</td>
</tr>
<tr>
<td>5.4</td>
<td>Reading Command Files</td>
</tr>
</tbody>
</table>
Abstract

The simulator described in this manual will interpret the output of the ISPL compiler, the RTM code, thus allowing the users a generalized computer architecture simulation facility. This manual describes the commands available to the users.

Acknowledgements

The ISP simulator is a much improved version of a primitive system developed by S. Rodkey at CMU during the spring of 1975. The system was modified and expanded by Greg Lloyd of the Naval Research Laboratory during the Fall of 1975. The system was further enhanced by the author during the Winter and Spring of 1976. Many commands and features were added to the system as part of the Army/Navy CFA project. Special thanks are due to the users of the system for their comments and suggestions, among them: H. Elovitz (NRL), R. Gordon (NUSC), R. Howbrigg (NUSC), D. Siewiorek (CMU), and S. Zuckerman (NRL).
1. Introduction

The ISPL compiler translates Computer Architecture (Instruction Set) Descriptions written in a subset of ISP [Bell71] into instructions for an idealized Register Transfer Machine (RTM) which can perform the primitive Register Transfer Operations needed to fetch, decode, and execute instructions. The ISP simulator is in effect an implementation of the Register Transfer Machine.

Some effort has been put into isolating the user from the low level detail of the RTM code. Under normal circumstances, the user will interact with the simulator using the names of registers, memories, procedures, etc, as declared in the ISPL description.

The simulator follows the convention of the ISPL compiler with regard to number representation, it uses an unsigned (pure magnitude) representation. Internally, the simulator uses multiple precision operations on the PDP-10 to execute the data operations and transfers. A current implementation limitation sets a limit of 140 bits for the length of the variables used in the register transfer operations (beware that the ISPL compiler will allow the user to declare registers and memories of arbitrary length - the simulator will warn the user if any attempt is made to operate on variables larger than 140 bits).

Although concurrency is easily described in ISPL, the simulator makes no attempt to provide this facility. It will execute concurrent operations in sequence and the user should avoid writing order-dependent parallel ISP statements.
2. From ISPL to RTM and Beyond

The process of obtaining a running simulator given a syntactically correct ISP description is rather simple. The ISPL compiler, in the absence of serious errors, will produce a MACRO10 program containing the RTM object code. This program should be assembled in order to produce a relocatable PDP-10 binary file. This process is also handled by the ISPL compiler (i.e. it will generate the RTM file and then invoke the MACRO10 assembler). At the end of the compilation the user has the following files (assume that the original ISP files was X.ISP):

- X.ISP (The source file)
- X.LST (The listing file, described in the ISPL compiler manual)
- X.RTM (The object file, described in the ISPL compiler manual)
- X.REL (The relocatable binary version of the X.RTM file)

At this point you can get rid of the X.LST and X.RTM files, as far as the simulator is concerned, they are not needed at all. Hold onto the X.REL file for dear life, unless cycles are cheap at your installation and you can afford to run the ISPL compiler as often as you please.

The simulator consists of a group of (currently 7) binary files that must be linked, using one of the standard PDP-10 CUSPs, with the X.REL file. Once this is done, you can save the core image and you are all set to go. The exact procedure might change from installation to installation, depending on whether you use LOAD or LINK10.

A typical procedure might look like:

EXECUTE X.REL@ISPSIM.CMD

<or alternative, if you have the LINK-10 loader in your system>

R LINK
*x.REL
@ISPSIM.CMD
<a/SSAVE x>

A-58
**/GO**

The above sequence will produce two files: X.SHR and X.LOW. These are your ISPL description compiled, linked, saved, and ready to run:

```plaintext
RU x < and off you go!, good luck!
```

RU x < and off you go!, good luck!
3. The Command Language

The simulator accepts a small number of commands, using a fixed format:

<keyword> <parameter> <parameter> ...

Only one command is accepted per line. Commands might be typed, in upper or lower case, directly from the user's terminal or can be retrieved from command files (the latter can be done recursively, up to 16 levels of nested command files). Comments can be inserted in the command stream by typing a "!" followed by any arbitrary string. The command scanner will ignore anything between the "!" and the end of the line. Most parameters represent ISPL variable names or numeric values. The latter can be typed in several modes (Binary, Octal, Decimal, and Hexadecimal) and there are facilities to set up a proper default value of the type-in type-out radix.

All variables, labels, and constants defined in the ISP source program have activity counters associated with them. This allows the user to collect statistical data when running benchmark programs under the simulator. There are commands to clear, preset, and interrogate the value of these counters.

The command language include a group of commands to trace variables, start, break, and continue a simulation run, as well as commands to set and interrogate the values of the register and memories of the target machine.

When the simulator is running and the user suspects an infinite loop of instructions, typing a $ (Altmode) will break the execution. Actually, any type ahead will produce an interruption. $ is the preferred mode.

3.1. START and CONTINUE

START <label> is the command used to begin the simulation of an ISP procedure or main program. <label> is the name of a procedure declared in the ISP description. The START command is valid only at the top level of simulation. Thus, after a breakpoint in the simulation the user must use the command CONT to proceed.

3.2. EXIT

EXIT is the command used to finish a simulation run. It allows an orderly return to the PDP-10 monitor. EXIT closes the files that might have been created with the OCONNECT command. Typing TC will return to the monitor but CONNECTed files will be lost.

3.3. READ and DUMP

READ <dev:filename.ext[ppn]> allows the user to specify a file containing simulation commands. Essentially, READ substitutes the user terminal with the file and proceeds to read and execute commands until the end of the file is found, at which point the user terminal is again the command input device. Defaults are DSK (device), SIM (extension) and current user's PPN. Command files can contain comments. A comment is anything between a ! and the end of a line.

DUMP is used to save the status of a simulation run. DUMP creates a file containing the values of each variable (if non-zero), trace/break flags, read/write counters, etc. The file created by DUMP can be read by the READ command, thus allowing a simple way of reinitializing a simulation at the point the DUMP command was issued.
3.4. ECHO and DECHO

ECHO and DECHO are commands used to set an internal flag that controls the
ECHOing of the commands being read from a command file onto the user terminal.
After the ECHO command is issued, the execution of a READ command will type onto
the user's terminal the command lines as they appear in the command file. DECHO
disables this type-out. ECHO and DECHO can be issued from inside the command file
thus allowing a selective type-out.

3.5. RADIX

RADIX <base> is used to set the numeric base to be used for typing in and out.<base> is one of the following strings: BINARY, OCTAL, DECIMAL, or HEX. If <base> is
omitted the command simply types the name of the current base without altering it.
The current base setting might be bypassed on input by prefixing the constant with
one of the following: ' (binary), * (octal) or " (hex). Regardless of the current radix,
HEX constants which begin with a letter MUST be prefixed with " (this is a requirement
that will be lifted in a future release).

3.6. CTR, SETCTR, and OUTCTR

CTR <name> displays the value of the counter(s) associated with <name>. These
counters are tagged with R, W, or L to indicate whether they are the Read, Write, or
Label count respectively. SETCTR <name> <readcounter> <writecounter> allows the
user to specify the setting of these counters. If <name> is a label, then the
<readcounter> plays the role of label count. If the <counter> values are omitted
they default to 0. Instead of <name> the user may specify ALL and the command is
applied to all the variables and labels. All read/write counts are expressed in terms of 8-bit bytes. Thus, reading a 16 bit register increments the R counter by 2. The register lengths are rounded up to the next multiple of 8 before incrementing the counter: A 19 bit register counts as 3.

OUTCTR <filename.ext[ppn]> is a subset of the DUMP command. It creates a file (default extension CTR) with the values of all non-zero counters.

3.7. OPAQUE and Dopaque

OPAQUE <label-list> and Dopaque <label-list> are used to inhibit or enable the variable and label activity counters. The parameters to these two commands are labels or procedure names. If a procedure is OPAQUEd then no activity counts are incremented during its execution. The Dopaque command re-enables the activity counting. These two commands affect only those procedures named in the parameter list. Procedures called by OPAQUEd or Dopaque procedures are not affected.

3.8. VALUE and SetValue

VALUE and SetValue are the commands used to set and interrogate the contents of the ISP variables. The valid formats are:

VALUE <regname> (displays the value of a single register)

VALUE <memname> [ <fromword> [: <toword> ] ] (displays the values stored in a memory).

SETVAL <regname> = <value> (stores <value> into the register)

SETVAL <memname> [ <fromword> ] = <value-list> (stores into the memory. If more than one value is specified, they are stored in successive memory positions, starting at <fromword>).

A-63
3.9. TRACE, UTRACE, DTRACE, and TELLTRace

{TRACE | UTRACE | DTRACE} <variable-list> are the commands used to enable or disable the tracing of variables during the simulation. If the identifier ALL is specified instead of a variable list, the command applies to all variables. TRACE and UTRACE differ in that the former applies to all variables (including compiler declared temporary registers and flags) while the latter only applies to user declared variables (registers and memories). DTRACE is used to disable the tracing.

TELLTRace will type on the user's terminal the list of variables currently being traced.

3.10. BREAK, DBREAK, and TELLBReak

{BREAK | DBREAK} <label-list> are the commands used to enable or disable the setting of breakpoints during the simulation. The parameters are either ISP procedure names or labels. TELLBR displays on the user's terminal the list of breakpoint names.

3.11. SBREAK, DSBREAK, and TELLSBreak

These commands are similar to BREAK, DBREAK, and TELLBReak but instead of using ISP labels as parameters they take RTM statement numbers. Thus allowing a finer degree of control on the placement of the breakpoints. These commands are not particularly useful for the normal user, who should not be concerned with the RTM code.

3.12. ICONNEct and OCONNEct

ICONNEct <identifier>,<channel-number>,<variable-name>

OCONNEct <identifier>,<channel-number>,<variable-name>
These commands are used to "connect" ISP variables to PDP-10 ASCII files which will act as potentially infinite sources/sinks for variable values. When a variable is connected to an input file, each time the variable is accessed, the value will be obtained from the file instead of the simulated storage allocated to the variable. Similarly, writing into a variable that has been connected to an output file results in the value being written into the file (as well as into the storage allocated to the variable). The format for both input and output files is the same: one number/line.

The file names are created by the simulator and consist of the first parameter to the command (the <identifier>) as the file name, with extension ICn (ICONNEct) or OCn (OCONNEct), where n is the user specified channel number. The current implementation only allows up to three input and three output channels open simultaneously. Thus the only valid channel numbers are 1, 2 and 3.

3.13. HELP

HELP tells the user about the command names and their format. HELP <commandname> tells the user about a specific command.
4. Storage Mapping

The simulator allocates space for the registers and memories declared in the RTM symbol table using contiguous storage on the memory of the PDP-10. The fact that the PDP-10 is a 36 bits/word, 2's complement machine is completely transparent to the user. All RTM operations are interpreted rather than compiled into PDP-10 instructions. Moreover, the simulator does not impose any limitations derived from the word length; ISPL registers and memories are allocated contiguous bit strings on the PDP-10.

The use of logical register/memory declarations in the ISPL description presents the following problem: The ISPL compiler allows the user to define arbitrary mappings between bits of the left and right hand sides of the logical declaration, the only check made at that point is that the number of bits is the same. From the simulator point of view, it could be possible to implement arbitrary bit mappings at a tremendous degradation in performance (accessing a bit of a register or memory word that is mapped onto some other component implies searching a table of bit name/position equivalences; having to follow this procedure bit by bit, even for full register/word accesses could be hard to justify). The simulator makes a compromise between convenience to the ISPL writer and efficiency of simulation. The solution adopted is to restrict the types of mappings that the simulator can handle: all the bits of the right hand side of a logical declaration must be contiguous. Continuity is defined in terms of the word/bit naming convention used in the main declaration of the register/memory used on the right hand side of the logical declaration. There are no limitations as to what can appear on the left hand side of the logical declaration, these bits are by definition contiguous.
Specifically, the following are the valid types of mappings allowed by the simulator:

1) If the right hand side of a mapping was declared as a register, the structure of the right hand side must specify a contiguous string of bit names as specified in the main declaration. The number of bits may range from 1 to the entire register length and, for proper subsets of the main declaration, may be located anywhere in the register.

2) If the right hand side of a mapping consists of a single memory word, the valid mappings are those defined as above.

3) If the right hand side of a mapping consists of a set of memory words, the structure of the right hand side must specify a contiguous string of full words as specified in the main declaration. The number of words may range from 1 to the entire memory range and, for proper subsets of the main declaration, may be located anywhere in the memory.
4.1. Allowable Types of Mapping

The following list of memory maps gives a good coverage of the allowable cases:

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M[77777:77888, 77777:0] &lt;7:0&gt;</td>
<td>THE ADDRESSING SPACE</td>
</tr>
<tr>
<td>MB[77777:0] &lt;7:0&gt;</td>
<td>= M[77777:0] &lt;7:0&gt;</td>
</tr>
<tr>
<td>MBO[77777:77888] &lt;7:0&gt;</td>
<td>= M[77777:77888] &lt;7:0&gt;</td>
</tr>
<tr>
<td>M10[77777:0] &lt;15:0&gt;</td>
<td>= M[77777:0] &lt;7:0&gt;</td>
</tr>
<tr>
<td>M10[77777:77888] &lt;15:0&gt;</td>
<td>= M[77777:77888] &lt;7:0&gt;</td>
</tr>
<tr>
<td>B0N0[0:255] &lt;0:15&gt;</td>
<td></td>
</tr>
<tr>
<td>B0N0[15:0] &lt;15:0&gt;</td>
<td></td>
</tr>
<tr>
<td>B0N0[255:0] &lt;0:15&gt;</td>
<td></td>
</tr>
<tr>
<td>B0N0[0:255] &lt;15:0&gt;</td>
<td></td>
</tr>
<tr>
<td>RN0[0:15]</td>
<td></td>
</tr>
<tr>
<td>RN0[15:0]</td>
<td></td>
</tr>
<tr>
<td>MAP1[0:15] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP2[15:0] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP3[0:15] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP4[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP5[0:15] &lt;15:0&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP6[15:0] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP7[15:0] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP8[15:0] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP9[0:15] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP10[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP11[0:15] &lt;15:0&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP12[15:0] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP13[0:2] &lt;0:2&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP14[0:2] &lt;0:2&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP15[0:15] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP16[15:0] &lt;15:0&gt;</td>
<td>= B0N0[100:115] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP17[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP18[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP19[0:15] &lt;0:15&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP20[0:15] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP21[0:15] &lt;15:0&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP22[15:0] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP23[0:15] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP24[15:0] &lt;0:15&gt;</td>
<td>= B0N0[100:115] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP25[0:2] &lt;0:2&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP26[0:2] &lt;0:2&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP27[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP28[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP29[0:15] &lt;0:15&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP30[0:15] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP31[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP32[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP33[0:15] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP34[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP35[0:15] &lt;0:15&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP36[15:0] &lt;0:15&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP37[0:15] &lt;15:0&gt;</td>
<td>= B0N0[100:115] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP38[15:0] &lt;15:0&gt;</td>
<td>= B0N0[100:115] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP39[0:15] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;0:15&gt;</td>
</tr>
<tr>
<td>MAP40[15:0] &lt;0:15&gt;</td>
<td>= B0N0[115:100] &lt;15:0&gt;</td>
</tr>
<tr>
<td>MAP41[0:15] &lt;15:0&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP42[15:0] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP43[0:15] &lt;15:0&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP44[15:0] &lt;15:0&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP45[0:2] &lt;0:2&gt;</td>
<td>= RN0[5:13]</td>
</tr>
<tr>
<td>MAP46[0:2] &lt;0:2&gt;</td>
<td>= RN0[13:5]</td>
</tr>
<tr>
<td>MAP47[0:5]</td>
<td>= B0N0[100] &lt;4:5&gt;</td>
</tr>
<tr>
<td>MAP48[0:5]</td>
<td>= B0N0[100] &lt;1:4&gt;</td>
</tr>
<tr>
<td>MAP49[0:5]</td>
<td>= RN0[5:10]</td>
</tr>
<tr>
<td>MAP50[0:5]</td>
<td>= RN0[10:5]</td>
</tr>
</tbody>
</table>
5. **Examples**

This section contains the transcript of several actual runs. The first example is based on the small ISPL example described in the ISPL manual. The transcript for the compilation phase of the multiplier example appears in the ISPL compiler manual. We start from the point right after the MACRO10 assembler has generated the $REL file.

5.1. Linking the Compiler Output with the Simulator

```
$ link
mult
@ispsim
#ssave mult
#go
EXIT
```

MULT.REL is the name of the file created by the ISPL compiler. ISPSIM.CMD is the name of the command file containing the list of files that make up the simulator. It also contains commands to load the BLISS10 run time library. The use of the SSAVE switch instead of the SAVE switch creates a shareable version of the program. Thus the result of the LINK10 execution will be named MULT.SHR+MULT.LOW.
5.2. Running the Simulator

Here we run the program that was created in the previous transcript. The example makes use of a few simple commands that set initial values in the variables, selects some variables for tracing and then starts the execution at the main entry point of the description. The example is simple and self explanatory.

```plaintext
ru mult
ISP SIMULATOR V3 - NAL ARF STAGE 2
Thursday 29 Jul 76 23:42:13 MULT.ISP(NS55MB25)
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

> radix octal
> setval p=2
> setval mpd=3800
> trace mpd,p,c
> start 10

@ L0  #2 C  #10
@ STEP +#4 P  #1
@ L1  +#4 C  #7
@ STEP +#10 P  #1400
@ L1  +#4 C  #6
@ STEP +#4 P  #600
@ L1  +#4 C  #5
@ STEP +#4 P  #300
@ L1  +#4 C  #4
@ STEP +#4 P  #1400
@ L1  +#4 C  #3
@ STEP +#4 P  #60
@ L1  +#4 C  #2
@ STEP +#4 P  #30
@ L1  +#4 C  #1
@ STEP +#4 P  #14
@ L1  +#4 C  #0

SIMULATION COMPLETED

RUN TIME (10 usec units)=45259
RTM OPS EXECUTED=130

> value p
P  #14
> value mpd
MPD  #3800
> exit
EXIT
```
When the simulator starts it performs two preliminary operations: 1) It transforms the RTM statement table eliminating the DIVERGE/PMERGE operations that define concurrent operations, and 2) It allocates space for the registers and memories declared in the RTM symbol table. The simulator then types two messages advising the user of the existence of the HELP command and of the use of the <ESC> (AltMode) to break the execution of the simulator from the user's terminal.

The tracing of variables indicates the place in the ISPL program where an assignment to the variable has occurred. The location is identified by printing the nearest ISPL label together with a displacement (in RTM operations) from this label. The name of the variable affected by the transfer is printed, together with the new value. The run time printed at the end of the simulation is obtained from a fast 10us clock available at CMU. Some installations might now have this feature.

In the above example we initialize the multiplier (P) to 2 and the multiplicand (MPD) to 6. According to the algorithm, the multiplicand is stored in the left half of the MPD register. In the current implementation of the simulator we can not specify partial register initialization, thus, we have to load the right half of MPD with a suitable value (initialization of variables in the command language implies full register modification, with zeroes on the left of the value). At the end of the run, the contents of the P register contains the result of the multiplication (6*2=12 or #14 given that we set the type out radix to OCTAL).
5.3. Executing Selected Procedures

In the following example we show a few more commands and features of the simulator:

```
ru mult
ISP SIMULATOR V3 - NRL ARF STAGE 2
Thursday 29 Jul 76 23:42:13 MULT.ISP(NS65MB25)
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

> radix octal
> setval p=3
> setval mpd=400 ! Multiplicand=1
> utrace all
> start step
```

```
@ STEP  #10 P =#201
RUN TIME(10 usec units)=3001
RTOS OPS EXECUTED=9
```

The above sequence shows how the simulator can be used to execute selected procedures from the ISPL description. In fact, the simulator treats ALL labels and procedure names as potential entry points. It does not assign any special meaning to the label of the main body of the ISPL description.
5.4. Reading Command Files

The following example shows the use of the READ command. In this particular case we are not only initializing the variables and setting trace flags, but we are also starting the simulation automatically from the command file. The number of "->" character used to prompt the input stream (a user or a command file) indicates the level of nesting of the command stream. One "->" is the mark of the top level.

> dtrace all
> read ml.sim
>> ! this is a command file
>> setval p=2
>> setval mp=2000
>> 1 multiplier=4
>> trace p
>> start 10
   @ STEP +#4 P  =#1
   @ STEP +#10 P =#1000
   @ STEP +#4 P  =#400
   @ STEP +#4 P  =#200
   @ STEP +#4 P  =#100
   @ STEP +#4 P  =#40
   @ STEP +#4 P  =#20
   @ STEP +#4 P  =#10
SIMULATION COMPLETED

RUN TIME (10 usec units)=32120
RTM OPS EXECUTED=136

>> land of command file
>> 7 LINES READ
> exit
EXIT
ISP DESCRIPTION OF the IBM S/360, Interdata 8/32, and DEC PDP-11

! ISP DESCRIPTION OF IBM SYSTEM/360 ARCHITECTURE

! THIS DESCRIPTION INCLUDES THE STANDARD INSTRUCTION SET ONLY.
! THE FLOATING-POINT FEATURE INSTRUCTIONS AND DECIMAL FEATURE
! INSTRUCTIONS ARE NOT DESCRIBED.
! THE PROTECTION FEATURE INSTRUCTIONS AND DIRECT CONTROL FEATURE
! INSTRUCTIONS ARE DESCRIBED.
! THE TEST AND SET INSTRUCTION IS NOT DESCRIBED DUE TO THE
! LIMITATIONS OF A SINGLE ISP DESCRIPTION TO COVER TWO INDEPENDENT
! PROCESSES. A SECOND PROCESS (ISP DESCRIPTION) SHOULD BE GIVEN
! FOR THE MEMORY LATCHING.
! THE DIAGNOSE INSTRUCTION DESCRIPTION ( WHICH IS A MODEL DEPENDENT
! INSTRUCTION ) WAS MODIFIED FOR USE AS A HALTING MECHANISM FOR
! THE SIMULATION. THEREFORE, THE DIAGNOSE INSTRUCTION DOES NOT
! CORRESPOND TO ANY S/360 MODEL DIAGNOSE INSTRUCTION.
! THE CLC ( COMPARE LOGICAL LONG ) INSTRUCTION FROM THE S/370
! WAS ADDED FOR RUNNING BENCHMARKS. IT IS NOT A TRUE DESCRIPTION
! OF THE INSTRUCTION SINCE IT IS NOT INTERRUPTABLE.
! ADDITIONAL LABELS WERE ADDED TO HELP IN MEASURING THE BENCHMARK
! PROGRAMS. THESE ARE NOT PART OF THE ARCHITECTURE DESCRIPTION.

S360:

(DECLARE

MACRO MAXIM:2047 $
MACRO MAXI:4095 $
MACRO MAXII:8191 $
MACRO MAXII:16383 $
MACRO MAXKEY:7 $
MACRO BEGIN:=( $)
MACRO END:=) $

! PRIMARY MEMORY

MEMD(0:MAXDU)<8:63>;
MEMW(0:MAXW)<0:31>:=MEMD(0:MAXDU)<8:63>;
MEMB(0:MAXB)<0:15>:=MEMD(0:MAXDU)<8:15>;
MEMH(0:MAXH)<0:7>:=MEMH(0:MAXDU)<8:7>;
MEMO(0:MAXO)<0:6>:=MEMD(0:MAXDU)<8:6>;
STKEYS(0:MAXKEY)<0:6>;

! STOGE KEY ARRAY

! PERMANENT STORAGE ASSIGNMENTS

IPLPSW<0:63>:=MEMD(10:71)<0:7>;
IPLPSW<0:63>:=MEMD(16:23)<0:7>;
IPLCH<0:63>:=MEMB(48:64)<0:8>;
IPLCH<0:63>:=MEMB(48:64)<0:8>;
EKPWSW<0:63>:=MEMD(72:79)<0:7>;
EKPWSW<0:63>:=MEMD(72:79)<0:7>;
EXPWSW<0:63>:=MEMD(72:79)<0:7>;
EXPWSW<0:63>:=MEMD(72:79)<0:7>;
SVNPSW<0:63>:=MEMD(96:103)<0:7>;
SVNPSW<0:63>:=MEMD(96:103)<0:7>;
PRMPSW<0:63>:=MEMD(112:119)<0:7>;
PRMPSW<0:63>:=MEMD(112:119)<0:7>;
MCNPSW<0:63>:=MEMD(112:119)<0:7>;
MCNPSW<0:63>:=MEMD(112:119)<0:7>;
IONPSW<0:63>:=MEMD(128:127)<0:7>;
IONPSW<0:63>:=MEMD(128:127)<0:7>;
SCMW<0:63>:=MEMD(128:135)<0:7>;
SCMW<0:63>:=MEMD(128:135)<0:7>;

! PROCESSOR STATE

REG0(0:15)<0:31>;
PSI<0:63>;
PSW(0:32)<0:15>:=PSW<0:63>;
CHANNEL<0:7>:=PSW<0:7>;
CHANNEL<0:7>:=PSW<0:7>;
PROT<0:3>:=PSW<0:11>;
PROT<0:3>:=PSW<0:11>;
ASCH<0:12>;
ASCH<0:12>;
MACM<0:13>;
MACM<0:13>;
WAIT<0:14>;
WAIT<0:14>;
PROB<0:15>;
PROB<0:15>;
INTC<0:15>:=PSW<10:31>;
INTC<0:15>:=PSW<10:31>;

B-1
<table>
<thead>
<tr>
<th>ISP Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILC&lt;0:1&gt;</td>
<td>Instruction Length Code</td>
</tr>
<tr>
<td>CC&lt;0:1&gt;</td>
<td>Condition Code</td>
</tr>
<tr>
<td>FPOMS&lt;0:1&gt;</td>
<td>Fixed Point Overflow Mask</td>
</tr>
<tr>
<td>DOPMS&lt;0:1&gt;</td>
<td>Decimal Overflow Mask</td>
</tr>
<tr>
<td>EXOFMS&lt;0:1&gt;</td>
<td>Exponent Underflow Mask</td>
</tr>
<tr>
<td>SIGMS&lt;0:1&gt;</td>
<td>Significance Mask</td>
</tr>
<tr>
<td>PC&lt;0:23&gt;</td>
<td>Program Counter (24 Bits)</td>
</tr>
</tbody>
</table>
IMPLEMENTATION RELATED VARIABLES

THESE DECLARATIONS AND DEFINITIONS ARE NOT ACTUALLY PART OF THE ARCHITECTURE DESCRIPTION, BUT ARE NECESSARY FOR THE ISP DESCRIPTION.

**Instruction Register**
- 1/2 Word Address for IN (IN Execute)
- RR, RX, RS, S1, SS
- RR, RX, RS
- RR
- RX
- RX, RS, S1, SS
- RX, RS, S1, SS
- RS
- MASK 1
- SI
- SS
- SS

**Memory Address Register**
- Left Byte in MBR
- Right Byte in MBR
- Memory Buffer Register
- Byte Count Register 1
- Byte Count Register 2
- 17 Bit Temporary
- 64 Bit Dividend Register
- Execute Recursion Flag
- Zone Temporary
- Digit Temporary
- Scale Factor for CVB
- No-Op Register
- 1 Bit Temporary
- 1 Bit Auxiliary Temp
- 2 Bit Temporary
- 2 Bit Auxiliary Temp
- 4 Bit Temporary
- 6 **
- 8 **
- 8 ** Auxiliary Temp
- 16 ** Temp
- 24 **
- 32 **
- 33 **
- 64 **

**Overflow**
- Stop Switch
- Interrupt Vector
- Bit 0 = Machine Check
- Bit 1 = SVC
- Bit 2 = Prog Check
- Bit 3 = External Interrupt (Timer)
- Bit 4 = I/O Interrupt
- Channel Mask Register
- Channel Release
- Channel Select Register
- Channel Condition Code
- Channel Instruction Line
- 0 => S10
- 1 => T10
- 2 => MID
- 3 => TCH

**Channel Address Register**
- Device Register
- Holds Device Address (0–255)
- External Register
- Bit 0 = Timer Interrupt
- Bit 1 = Console Interrupt
S380 ISP DESCRIPTION

100NEG<0:7>;

SIGOUT<0:5>;

MACRO NOP:TB-0 $
S360 ISP DESCRIPTION

I UTILITY ROUTINES

I PRIVILEGED STATE CHECK ROUTINE

PSCK:
BEGIN
IF PROBST => INTCODE-2; INTVEC<2>=1 NEXT BAILOUT ICYCLE
END;

I INTERRUPT CODE 5 IMPLIES ADDRESSING ERROR
I INTERRUPT CODE 6 IMPLIES SPECIFICATION (ALIGNMENT ERROR)
I INTERRUPT CODE 4 IMPLIES PROTECTION
I THE ORDER OF SETTING THESE CODES MAY BE IMPLEMENTATION DEPENDENT
I TESTS ON A MODEL 75 SHOWN CODE 6 IS FIRST

I CHECK ROUTINE FOR STORAGE PROTECTION

CKPR:
BEGIN
IF STREY(MAR<0:12>)-1<1:6> MON PR0TE0Y =>
INTVE5<2>-1; INTCODE-4 NEXT BAILOUT ICYCLE
END;  I END OF CKPR

I CHECK ROUTINE FOR READ PROTECTION

CKRPR:
BEGIN
IF STREY(MAR<0:12>);<6> => CKPR
END;  I END OF CKRPR

I CHECK ROUTINE FOR 8SK & ISK INSTRUCTIONS

KEYCK:
BEGIN
PSCK NEXT
(IF REGfR2<2>8:16> => INTVEC<2>=1; INTCODE-8 NEXT BAILOUT ICYCLE) NEXT
(IF REG<21:23> GTR MAR KEY => INTVEC<2>=1; INTCODE-8 NEXT BAILOUT ICYCLE)
END;  I END OF KEYCK

I CHECK ROUTINE FOR BYTE ADDRESSES

CKBTR:
BEGIN
IF MAR GTR MARB => INTCODE-5; INTVEC<2>=1 NEXT BAILOUT ICYCLE
END;

I CHECK 1/2 WORD ADDRESS ROUTINE

CKHW:
BEGIN
IF MAR<23> => INTCODE-6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
(IF MAR<8:21> GTR MAH => INTCODE-5; INTVEC<2>=1 NEXT BAILOUT ICYCLE)
END;  I END OF CKHW

I CHECK WORD ADDRESS ROUTINE

CKW:
BEGIN
(IF MAR<22:23> => INTCODE-6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
(IF MAR<8:21> GTR MAH => INTCODE-5; INTVEC<2>=1 NEXT BAILOUT ICYCLE)
END;  I END OF CKW

I CHECK DOUBLE WORD ADDRESS ROUTINE

CKDW:
BEGIN
(IF MAR<21:23> => INTCODE-8; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
(IF MAR<8:20> GTR MAH => INTCODE-5 INTVEC<2>=1 NEXT BAILOUT ICYCLE)
END;  I END OF CKDW

I READ A BYTE ROUTINE

ROBYTE:
S360 ISP DESCRIPTION

BEGIN
CKBTAD NEXT
CKDPR NEXT
MBR<24:31>=MEMMAR
END; ! END OF RBYTE

! WRITE A BYTE ROUTINE

WRBYTE:=
BEGIN
CKBTAD NEXT
CKDPR NEXT
MBR(MAR)=MBR<24:31>
END; ! END OF WRBYTE

! READ A 1/2 WORD ROUTINE

READW:=
BEGIN
CKWHAD NEXT
CKDPR NEXT
MBR<31:16>=MEMMAR<16:22>
END; ! END OF READW
S360 ISP DESCRIPTION

! WRITE A 1/2 WORD ROUTINE

WRITE:=
BEGIN
CKWORD NEXT
CKPR NEXT
MENH(MAR<0:22>)=MBR<16:31>
END; \ END OF WRITE

! READ A WORD ROUTINE

READ2:=
BEGIN
CKWORD NEXT
CKPR NEXT
MBR=MENH(MAR<0:21>)
END; ! END OF READ2

! WRITE A WORD ROUTINE

WRITE:=
BEGIN
CKWORD NEXT
CKPR NEXT
MENH(MAR<0:21>)=MBR
END; \ END OF WRITE

! OPERAND ONE ADDRESSING FOR SS

ADDRESS1:=
BEGIN
MAR=(MAR1+LAUX1)<23:0>
END;

! OPERAND TWO ADDRESSING FOR SS

ADDRESS2:=
BEGIN
MAR=(MAR2+LAUX2)<23:0>
END;

! FETCH OF L2 OPERAND IF POSSIBLE OR A LOAD OF ZERO INTO THE MBA IF L2 FIELD IS EXHAUSTED

L2FETCH:=
BEGIN
DECODE (LAUX2 EQL 0) =>
\0 BEGIN
LAUX2=(LAUX2 MINUS 1)<7:0> NEXT
ADDRESS2 NEXT ADDRESS
}\1 MBR<0>
END; \ END OF L2FETCH

! DEVICE ADDRESSING FOR I/O INSTRUCTIONS

ADDRESS:=
BEGIN
CHAREG=(D1<REG1>)<15:8>
DEVREG=(D1<REG1>)<7:0>
END;

! SIGN EXTENSION HALFWORD TO FULLWORD IN MBA

SIGNEXT:=
BEGIN
DECODE MBR<15>=
\0 MBR<0:15>="0000"
\1 MBR<0:15>="FFFF"
END; \ END OF SIGNEXT
S360 ISP DESCRIPTION

! SET FIXED POINT CONDITION CODES

SETFCC:
BEGIN
CC=0 NEXT
(DECODE REG(1)<0>>
\0  (IF REG(1)==CC<2>)
\1  (CC=1)
NEXT
(IF OVF => CC=3) NEXT
(IF OVF AND FTPPM => INTVEC<2+1>; INTCOE=0 NEXT Bailout ICYCLE)
END;  ! END OF SETFCC

! ILLEGAL OP-CODE

OPEK:
BEGIN
INTCOE=1; INTVEC<2+1 NEXT Bailout ICYCLE
END;

! INSTRUCTION FETCH ROUTINE

IFETCH:
BEGIN
MAR=PC.Next READIN NEXT
IR<8:15>=MBR<16:31>-;
ILC=MBR<16>=MBR<17>=1;
Pc=PC=(MBR<16>=MBR<17>=1>[02]<23:8>;
OVF=0 NEXT
(IF ILC GTR 1 =>
  MAR=(MAR+2)<23:0> NEXT
  READIN NEXT
  IR<16:31>=MBR<16:31> NEXT
  (IF ILC GTR 2 =>
    MAR=(MAR+2)<23:0> NEXT
    READIN NEXT
    IR<32:47>=MBR<16:31>
  )); ! END OF IF ILC GTR 1
) ! END OF IFETCH
S360 ISP DESCRIPTION

**PR INSTRUCTIONS**

**S360**

SET PROGRAM MASK
BEGIN
PSW<14:39>=REG(R1)<2:7>
END; /* END OF S360 */

**BALR**

BRANCH AND LINK REGISTER
BEGIN
T24=REG(R2)<8:31> NEXT
REG(R1)=PSW<32:63> NEXT
(IF R2 =>
  BALR1:=(PC+T24)
)
END; /* END OF BALR */

**BCTR**

BRANCH ON COUNTER REGISTER
BEGIN
T24=REG(R2)<8:31> NEXT
REG(R1)=(REG(R1) MINUS 1)<31:0> NEXT
(IF REG(R1) =>
  (IF R2 =>
    BCTR1:=(PC+T24)
  ) /* END OF IF R2 */
)
END; /* END OF BCTR */

**BCR**

BRANCH ON CONDITION REG
BEGIN
IF M1CC =>
  (IF R2 =>
    BCR1:=(PC+REG(R2)<8:31>)
)
END; /* END OF BCR */

**SSK**

SET STORAGE KEY (PROTECTION FEATURE INSTRUCTION)
BEGIN
KEYCK NEXT
STKEYS(REG(R2)<8:20>)=REG(R1)<24:28>
END; /* END OF SSK */

**ISK**

INSERT STORAGE KEY (PROTECTION FEATURE INSTRUCTION)
BEGIN
KEYCK NEXT
REG(R1)<24:28>=STKEYS(REG(R2)<8:20>) NEXT
REG(R1)<20:31>=0 NEXT
END; /* END OF ISK */

**SVC**

SUPERVISOR CALL
BEGIN
INTCODE=12; INTVEC=1 NEXT BAILOUT ICYCLE
END;

**CLCL**

COMPARE LOGICAL LONG (5/370)

THIS INSTRUCTION WAS ADDED FOR THE RUNNING OF
BENCHMARK PROGRAMS. IT IS NOT A TRUE DESCRIPTION
OF THE INSTRUCTION SINCE IT IS NOT INTERRUPTABLE
IN ITS PRESENT FORM.
BEGIN
(IF R1<3 OR R2<3) => INTCODE=6; INTVEC=2 NEXT BAILOUT ICYCLE) NEXT
CLCL1:=(CC<8>) NEXT
CLCL1: BEGIN
IF (REG(R1)<1)<8:31> NEQ 0 OR (REG(R2)<1)<8:31> NEQ 0) =>
  (DECODE (REG(R1)<1)<8:31> NEQ 0) =>
    \0
    TB-REG(R1+1)<8:17>
    \1 BEGIN
    MAR-REG(R1)<8:31> NEXT
    RBYTE NEXT
    TB-MBA<24:31>
    END /* END OF \
    ) NEXT /* END OF DECODE
  (DECODE (REG(R2)<1)<8:31> NEQ 0) =>

B-9
S360 ISP DESCRIPTION

\0 T8A-REG(R2+1)<0:7>1
\1 BEGIN
MAR-REG(R2)<8:31> NEXT
ROBYTE NEXT
T8A-MBS<24:31>
END  END OF \1
) NEXT  END OF DECODE

CLCLC2: BEGIN
DECODE T8 TST T8A =>
\SS  CC-1
\EQL CC-2
\GTR CC-2
END NEXT  END OF CLCLC2

(IF CC EQL 0 =>
(1F REG(R1+1)<8:31> NEQ 0 =>
REG(R1+1)=<REG(R1)+1><23:0>
REG(R1+1)<8:31>=<REG(R1)+1> MINUS 1)<23:0>
) NEXT  END OF IF REG(R1+1)

(IF REG(R2+1)<8:31> NEQ 0 =>
REG(R2+1)=(REG(R2)+1)<23:0>
REG(R2+1)<8:31>=<REG(R2)+1> MINUS 1)<23:0>
) NEXT  END OF IF REG(R2+1)

CLCL1)  END OF IF CC

END;  END OF CLCL
SUB ISP

LPR:= LOAD POSITIVE REGISTER
BEGIN
(DECODEREG(R1)<0> =>
\0REG(R1)<REG(R2)>
\1REG(R1)<MINUS REG(R2)<31:0>
) NEXT
OVF<(REG(R1)) EQL "80000000" NEXT
SETFCC
END; ! END OF LPR

LNR:= LOAD NEGATIVE REGISTER
BEGIN
(DECODEREG(R1)<0> =>
\0REG(R1)<MINUS REG(R2)<31:0>
\1REG(R1)<REG(R2)>
) NEXT
SETFCC
END; ! END OF LNR

LTR:= LOAD AND TEST REGISTER
BEGIN
REG(R1)<REG(R2) NEXT SETFCC
END;

NR:= AND REGISTER
BEGIN
REG(R1)<REG(R2) AND REG(R2) NEXT
NRCC:= BEGIN
CC<> NEXT
(IF REG(R1) => CC<-)
END ; ! END OF NRCC
END; ! END OF NR

CLR:= COMPARE LOGICAL REGISTER
BEGIN
CLRCC:= BEGIN
CC<> NEXT
(IF REG(R1) LSS REG(R2) =>
CC<-) NEXT
(IF REG(R1) GTR REG(R2) =>
CC<)
END ; ! END OF CLRCC
END; ! END OF CLR

OR:= OR REGISTER
BEGIN
ORCC:= BEGIN
CC<> NEXT
(IF REG(R1) OR REG(R2) =>
CC<-) NEXT
END ; ! END OF ORCC
END; ! END OF OR

XR:= EXCLUSIVE OR REGISTER
BEGIN
XRCC:= BEGIN
CC<> NEXT
(IF REG(R1) => CC<-)
END ; ! END OF XRCC
END; ! END OF XR
S360 ISP DESCRIPTION

LCR:= I LOAD AND COMPLEMENT REGISTER
BEGIN
REG(R1) = (MINUS REG(R2))<31:0> NEXT
OVF = (REG(R1) EQL "80000000") NEXT
SETFCC
END; ! END OF LCR

LR:= I LOAD REGISTER
BEGIN
REG(R1)=REG(R2)
END;

CR:= I COMPARE REGISTER
BEGIN
T33=(REG(R1) MINUS REG(R2)) NEXT
CRCC:= BEGIN
CC=0 NEXT
(IF T33 > CC=NOT T33<1> + 1)
END; ! END OF CRCC
END; ! END OF CR

AR:= I ADD REGISTER
BEGIN
T33=REG(R1)+REG(R2) NEXT
(DECODE REG(R1)<8>REG(R2)<8> ==
\00 OVF=(T33<8> NEQ T33<1>);
\01 NOP;
\10 NOP;
\11 OVF=(T33<8> NEQ T33<1>)
) NEXT ! END OF DECODE
REG(R1)=T33<1:32> NEXT
SETFCC
END; ! END OF AR

SR:= I SUBTRACT REGISTER
BEGIN
T33 = ((NOT REG(R2)) + REG(R1) + 1)<32:0> NEXT
(DECODE REG(R1)<8>REG(R2)<8> ==
\00 NOP;
\01 OVF = (T33<8> NEQ T33<1>);
\10 OVF = (T33<8> NEQ T33<1>);
\11 NOP
) NEXT ! END OF DECODE
REG(R1) = T33<1:32> NEXT
SETFCC
END; ! END OF SR
S360 ISP DESCRIPTION

MR:  
! MULIPLY REGISTER
BEGIN
(IF R1<3> == INTCOE-6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
T1=REG(R2)<0> XOR REG(R<1>+1)<0> NEXT
(DECOREG REG(R2)<0> ==
\0
T64-REG(R2);
\1
T64-(MINUS REG(R2))<31:0>
) NEXT
(DECOREG REG(R<1>+1)<0> ==
\0
T32=REG(R<1>);
\1
T32-(MINUS REG(R<1>))<31:0>
) NEXT
T64=T64<32:63>=T32 NEXT
(IFE1 == T64=(MINUS T64)<63:0>) NEXT
REG(R1)<0>=T64<32:63>
END;  ! END OF MR

DR:  
! DIVIDE REGISTER
BEGIN
(IF R1<3> == INTCOE-6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
T1=REG(R1)<0> XOR REG(R<2>><0>) ; T1=REG(R<1><0>); T32=REG(R2) NEXT
DlVREG=(REG(R1) REG(R<1>)) NEXT
(IF REG(R2)<0> ==
\0
T32=(MINUS T32)<31:0>
; (IF REG(R<1><0> ==
\0
DlVREG=(MINUS DlVREG)<63:0>) NEXT
(IF (DlVREG/T32)<63:31> == INTCOE-9; INTVEC<2>=1 NEXT BAILOUT ICYCLE)) NEXT
REG(R<1>)-=(DlVREG/T32)<31:0>
(IFE1 == REG(R<1>)-(MINUS REG(R<1>))<31:0>) NEXT
(IFE1 == REG(R<1>)-(MINUS REG(R<1>))<31:0>)
END;  ! END OF DIVIDE REGISTER

ALR:  
! ADD LOGICAL REGISTER
BEGIN
T33=REG(R1)+REG(R2) NEXT
REG(R1)=T33<1:32> NEXT
ALRCC== BEGIN
CC=T33<0>== (T33<1:32> NEQ 0)
END;  ! END OF ALRC

SLR:  
! SUBTRACT LOGICAL REGISTER
BEGIN
T33=((NOT REG(R2)) + REG(R1) + 1)<32:0> NEXT
REG(R1)=T33<1:32> NEXT
SLRCC== BEGIN
CC=T33<0>== (T33<1:32> NEQ 0)
END;  ! END OF SLRC

END;  ! END OF B-13
**S368 ISP DESCRIPTION**

**LPDR**: ! LOAD POSITIVE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LPDR

**LNDR**: ! LOAD NEGATIVE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LNDR

**LTOR**: ! LOAD AND TEST (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LTOR

**LCOR**: ! LOAD COMPLEMENT (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LCOR

**LNO**: ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LNO

**LRD**: ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LRD

**LMA**: ! LOAD UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LMA

**LSD**: ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LSD

**LMR**: ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LMR

**LDR**: ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LDR

**LCO**: ! LOAD COMPLEMENT (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LCO

**LNO**: ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LNO

**LSD**: ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LSD

**LMR**: ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LMR
S360 ISP DESCRIPTION

LTER:  ! LOAD AND TEST (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF LTER

LCER: ! LOAD COMPLEMENT (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF LCER

HER:  ! HALF (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF HER

LER:  ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF LER

CER:  ! COMPARE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF CER

AER:  ! ADD NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF AER

SER:  ! SUBTRACT NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF SER

MER:  ! MULTIPLY (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF MER

DER:  ! DIVIDE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF DER

AUR:  ! ADD UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF AUR

SUR:  ! SUBTRACT UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
N0P
END; 1 END OF SUR
### S368 ISP Description

#### RR Instruction Decode Table

```plaintext
BEGIN
  D3 OPCODE<2:7> =>
  OPEK; 00
  OPEK; 01
  OPEK; 02
  OPEK; 03
  OPEK; 04
  OPEK; 05
  OPEK; 06
  OPEK; 07
  OPEK; 08
  OPEK; 09
  OPEK; 0A
  OPEK; 0B
  OPEK; 0C
  OPEK; 0D
  OPEK; 0E
  CLKL; 0F COMPARE LOGICAL LONG (S/370)
  LPR; 10 LOAD POSITIVE
  LNR; 11 LOAD NEGATIVE
  LTR; 12 LOAD AND TEST
  LCR; 13 LOAD COMPLEMENT
  NA; 14 AND
  CLR; 15 COMPARE LOGICAL
  OA; 16 OR
  OA; 17 EXCLUSIVE OR
  LA; 18 LOAD
  CA; 19 COMPARE
  RA; 1A ADD
  SA; 1B SUBTRACT
  MA; 1C MULTIPLY
  DA; 1D DIVIDE
  ALR; 1E ADD LOGICAL
  SLR; 1F SUBTRACT LOGICAL
  LPOR; 20 LOAD POSITIVE (LONG)
  LNR; 21 LOAD NEGATIVE (LONG)
  LTR; 22 LOAD AND TEST (LONG)
  LCR; 23 LOAD COMPLEMENT (LONG)
  LA; 24 HALVE (LONG)
  OPEK; 25
  OPEK; 26
  OPEK; 27
  LDR; 28 LOAD (LONG)
  CDR; 29 COMPARE (LONG)
  ADR; 2A ADD NORMALIZED (LONG)
  SDR; 2B SUBTRACT NORMALIZED (LONG)
  MDR; 2C MULTIPLY (LONG)
  DOR; 2D DIVIDE (LONG)
  AVR; 2E ADD UNNORMALIZED (LONG)
  SWR; 2F SUBTRACT UNNORMALIZED (LONG)
  LPER; 30 LOAD POSITIVE (SHORT)
  LNER; 31 LOAD NEGATIVE (SHORT)
  LTER; 32 LOAD AND TEST (SHORT)
  LCR; 33 LOAD COMPLEMENT (SHORT)
  HER; 34 HALVE (SHORT)
  OPEK; 35
  OPEK; 36
  OPEK; 37
  LER; 38 LOAD (SHORT)
  CER; 39 COMPARE (SHORT)
  AER; 3A ADD NORMALIZED (SHORT)
  SER; 3B SUBTRACT NORMALIZED (SHORT)
  MER; 3C MULTIPLY (SHORT)
  DER; 3D DIVIDE (SHORT)
  ARE; 3E AND UNNORMALIZED (SHORT)
  SUR; 3F SUBTRACT UNNORMALIZED (SHORT)
END; END OF DECODE
```

---

**END; END OF RR**

---

B-16
S360 ISP DESCRIPTION

1 RX INSTRUCTIONS

STH: = STORE HALFWORD
BEGIN
MR=REG(R1)<16:31> NEXT WMAH END;

LA: = LOAD ADDRESS
BEGIN
REG(R1)=MAR END;

STC: = STORE CHARACTER
BEGIN
HIBYTE=REG(R1)<24:31> NEXT
WBYTE END;

IC: = INSERT CHARACTER
BEGIN
ROBYTE NEXT
REG(R1)<24:31>=HIBYTE END; END OF IC

EX: = EXECUTE
BEGIN
(IF EXRF => INTVEC<2>=1; INTCDE<3 NEXT BAILOUT ICYCLE) NEXT
T4=R1; T2=0 NEXT
EX1:= BEGIN
REOHH NEXT
INPUT(2)=MR<16:31> NEXT
T2=(T2+1)<1:0>; MAR=(MAR+2)<23:0> NEXT
(IF (IR<8><IR<13>) GE Q T2 => EX1) NEXT
(IF T4 => IR<8:15><IR<8:15> ON REG(T4)<24:31>) NEXT
EXRF EE1 END; END OF EX1 END; END OF EX

BAL: = BRANCH AND LINK
BEGIN
REG(R1)=PSW<32:63> NEXT
PC=MAR END;

BCT: = BRANCH ON COUNT
BEGIN
REG(R1)=(REG(R1) MINUS 1)<31:0> NEXT
(IF REG(R1) =>
BCT1:= (PC+MAR)
) END OF IF REG(R1)
) END OF BCT
BC: = ! BRANCH ON CONDITION
BEGIN
IF M1(CCI) =>
   BC1: = (PC - MAR)
END;

LH:= ! LOAD HALFWORD
BEGIN
READHW NEXT SGNEXT NEXT
REG(R1) = MBR
END; ! END OF LH

CH: = !COMPARE HALFWORD
BEGIN
READHW NEXT SGNEXT NEXT
T33 = REG(R1) MINUS MBR NEXT
BEGIN
CC: = CC=0 NEXT
(IF T33 => CC=NOT T33<1> + 1)
END; ! END OF CHCC
END; ! END OF CH

AH: = !ADD HALFWORD
BEGIN
READHW NEXT SGNEXT NEXT
T33 = REG(R1) + MBR NEXT
BEGIN
T33 = REG(R1) <Q> & MBR<0> =>
   \00 OVF = (T33<8> NEQ T33<1>);
   \01 NOP;
   \10 OVF = (T33<8> NEQ T33<1>);
   \11 OVF = (T33<8> NEQ T33<1>);
   ) NEXT ! END OF DECODE
REG(R1) = T33<1:32> NEXT
SETFCC
END; ! END OF AH

SH: = !SUBTRACT HALFWORD
BEGIN
READHW NEXT SGNEXT NEXT
T33 = (NOT MBR) + REG(R1) + 1<32:0> NEXT
BEGIN
T33 = REG(R1) <Q> & MBR<0> =>
   \00 OVF = (T33<8> NEQ T33<1>);
   \01 OVF = (T33<8> NEQ T33<1>);
   \10 OVF = (T33<8> NEQ T33<1>);
   \11 NOP
   ) NEXT ! END OF DECODE
REG(R1) = T33<1:32> NEXT
SETFCC
END; ! END OF SH
S368 ISP DESCRIPTION

MH:=
! MULTIPLY HALFWORD
! MULTIPLIER IN MBR, MULTIPLICAND IN R1
BEGIN
READH  NEXT  SGX NEXT
T1=REG(R1)<8> XOR MBR<8>  NEXT
( IF REG(R1)<8> => REG(R1)>=MINUS REG(R1)<31:8>)
( IF MBR<8> => MBR<MINUS MBR<31:8>) NEXT
REG(R1)=REG(R1)-M BR<31:8> NEXT
( IF T1 => REG(R1)>=MINUS REG(R1)<31:8>)
END;  ! END OF MH

CVD:=
! CONVERT TO DECIMAL
BEGIN
CKD AWK NEXT
MAR=(MAR+7)<23:8;
T1=REG(R1)<8>;
T32=REG(R1)<8> NEXT
IF T1 = T32=MINUS T32<31:8> NEXT
(DECOD E ASCIIenk=)
\0  (DECOD E T1 =>
\0  MBR<28:31>="11001101"
\1  MBR<28:31>="11011101"
) NEXT  ! END OF DECO D ASCIIEN
MBR<24:27>=T32 MINUS (M BR<31:0>)<3:0> NEXT
WBYTE;
T4=7 NEXT
CVD1:= BEGIN
IF T4 =>
  MAR=(MAR-MINUS 1)<23:8>
  T32-T32<10><31:8> NEXT
  M BR<28:31>=T32 MINUS (M BR<31:0>)<3:0> NEXT
  T32-T32<10><31:8> NEXT
  MBR<24:27>=T32 MINUS (M BR<31:0>)<3:0> NEXT
  WBYTE;
  T4=T4<3:0> NEXT
CVD1
END;  ! END OF CVD1

END;  ! END OF CVD

CVD:=
! CONVERT TO BINARY
BEGIN
T64=8; T4=0; SCALE=1000000000000000; CKD AWK NEXT
CVD1:= BEGIN
WBYTE NEXT
MAR=(MAR-1)<23:8>;
T1=0;
DIGIT=MBR<24:27> NEXT
CVD2:= BEGIN
( IF DIGIT GTR 9 =>
  IN TVEC<2>=1; IN TCOE<7 NEXT BAILOUT ICYCLE) NEXT ! END OF IF DIGIT
T64=(T64+(DIGIT*SCALE))<63:8> NEXT
( IF T4 LSS 14 =>
  SCALE=SCALE/10;
  T4=T4<4:32> NEXT
  ( DECO D T1 EQL 0 =>
  \0  CVD1;
  \1  BEGIN
  T1=1;
  DIGIT=MBR<28:31> NEXT
  CVD2
  END ! END OF \1
 ) ! END OF DECO D
 ) NEXT ! END OF IF T4
DIGIT=MBR<28:31> NEXT  ! SIGN
( IF DIGIT LSS 10 =>
  IN TVEC<2>=1; IN TCOE<7 NEXT BAILOUT ICYCLE) NEXT
( IF T64<8:32> =>
  IN TVEC<2>=1; IN TCOE<8 NEXT BAILOUT ICYCLE) NEXT
( IF (DIGIT EQL '1011) OR (DIGIT EQL '1101) =>
  T64=MINUS T64<63:8> NEXT
B-19
S360 ISP DESCRIPTION

REG(R1)-T64<32:63>
END; END OF CVB2
END; END OF CVB1
END; END OF CVB

ST: = ! STORE
BEGIN
MBA=REG(R1) NEXT
END; END OF ST

N: = ! AND
BEGIN
REG=REG(R1) NEXT
REG=REG(R1) AND MBA NEXT
END; END OF N

CL: = ! COMPARE LOGICAL
BEGIN
CLCC:= BEGIN
CC:= REG(R1) NEXT
IF REG(R1) GTR MBA => CC-2) NEXT
IF REG(R1) LS5 MBA => CC-1) END; END OF CLCC
END; END OF CL

O: = ! OR
BEGIN
OCC:= BEGIN
CC:= REG(R1) NEXT
IF REG(R1) LS5 MBA => CC-0) END; END OF OCC
END; END OF O
5360 ISP DESCRIPTION

X:= ! EXCLUSIVE OR
BEGIN
READWD NEXT
REG(R1)+REG(R1) XOR MBR NEXT
XCC:= BEGIN
 CC= REG(R1) NEQ 0
 END ! END OF XCC
END; ! END OF X

L:= ! LOAD
BEGIN
READWD NEXT
REG(R1)+MBR
END; ! END OF L

C:= ! COMPARE
BEGIN
READWD NEXT
T33= (REG(R1)) MINUS MBR) NEXT
CCC:= BEGIN
 CC= NOT NEXT
 (IF T33 => CC= NOT T33<1> + 1)
 END ! END OF CCC
END; ! END OF C

A:= ! ADD
BEGIN
READWD NEXT
T33= REG(R1)+MBR NEXT
(DECODEREG(R1)<0>MVR<8> ==
 \00 OV= (T33<8> NEQ T33<1>)
 \01 NOP;
 \10 NOP;
 \11 OV= (T33<8> NEQ T33<1>)
) NEXT ! END OF DECODE
REG(R1)=T33<1:32> NEXT
SETFCC
END; ! END OF A

S:= ! SUBTRACT
BEGIN
READWD NEXT
T33= (NOT MBR) + REG(R1) + 1<32:8> NEXT
(DECODEREG(R1)<0>MVR<8> ==
 \00 NOP;
 \01 OV= (T33<8> NEQ T33<1>)
 \10 OV= (T33<8> NEQ T33<1>)
 \11 NOP
) NEXT ! END OF DECODE
REG(R1)=T33<1:32> NEXT
SETFCC
END; ! END OF S
S360 ISP DESCRIPTION

M:
! MULTIPLY
BEGIN
(If R1<3 => INTCOE<6; INTVEC<2:1 NEXT BAILOUT ICYCLE) NEXT
READT NEXT
T1=(REG(R1+1)<8> XOR MBR<8>) NEXT
(If MBR<8> => MBR-(MINUS MBR)<31:8>)
(DECODE REG(R1+1)<8> =
\$
T32-\text{REG}(R1+1);
\$
T32-(\text{MINUS REG}(R1+1))<31:8>
) NEXT ! END OF DECODE
T64-T32@MBR NEXT
(If T1 => T64-(\text{MINUS T64})<63:8>) NEXT
REG(R1)=T64<0:31>;
REG(R1+1)=T64<32:63>
END; ! END OF M

D:
! DIVIDE
BEGIN
(If R1<3 => INTCOE<6; INTVEC<2:1 NEXT BAILOUT ICYCLE) NEXT
READT NEXT
DIVREC=\text{REG}(R1)<8> REG(R1+1);
T1=(\text{REG}(R1)<8> XOR MBR<8>) NEXT
(If DIVREC<8> =>
DIVREC-(\text{MINUS DIVREC})<63:8>)
(If MBR<8> =>
MBR-(\text{MINUS MBR})<31:8>) NEXT
(If (DIVREC/MBR)<63:31> => INTCOE<9; INTVEC<2:1 NEXT BAILOUT ICYCLE) NEXT
REG(R1+1)=(DIVREC/MBR)<31:8> NEXT
(If T1 => REG(R1+1)<(\text{MINUS REG}(R1+1))<31:8>) NEXT
T1-\text{REG}(R1)+\text{MBR}<31:8> NEXT
REG(R1)=\text{DIVREC MINUS((REG}(R1+1)\text{MBR})<31:8>) NEXT
(If T1 => REG(R1)<(\text{MINUS REG}(R1))<31:8>)
END; ! END OF D

A:
! ADD LOGICAL
BEGIN
READT NEXT
T33-\text{REG}(R1) MBR NEXT
REG(R1)+T33<1:32> NEXT
ALCC:= BEGIN
CC-T33<0:8>(T33<1:32> NEQ 0)
END; ! END OF ALCC
END; ! END OF A

S:
! SUBTRACT LOGICAL
BEGIN
READT NEXT
T33=((\text{NOT MBR}) + \text{REG}(R1)+1)<32:8> NEXT
REG(R1)+T33<1:32> NEXT
SLCC:= BEGIN
CC-T33<0:8>(T33<1:32> NEQ 0)
END; ! END OF SLCC
END; ! END OF SL

B-22
S360 ISP DESCRIPTION

STD:= ! STORE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF STD

LD:= ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LD

CD:= ! COMPARE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF CD

AD:= ! ADD NORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF AD

SD:= ! SUBTRACT NORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF SD

MD:= ! MULTIPLY (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF MD

DO:= ! DIVIDE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF DO

AW:= ! ADD UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF AW

SW:= ! SUBTRACT UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF SW

STE:= ! STORE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF STE

LE:= ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF LE

CE:= ! COMPARE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF CE

AE:= ! ADD NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF AE

SE:= ! SUBTRACT NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF SE

ME:= ! MULTIPLY (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
BEGIN
NOP
END; ! END OF ME
DE: \* DIVIDE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
   BEGIN
   NOP
   END; \* END OF DE

AU: \* ADD UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
   BEGIN
   NOP
   END; \* END OF AU

SU: \* SUBTRACT UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
   BEGIN
   NOP
   END; \* END OF SU
RX INSTRUCTION DECODE TABLE

BEGIN
MAR:= 0;
NEXT
<table>
<thead>
<tr>
<th>EFFECTIVE ADDRESS CALCULATION FOR RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX:</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

(IF B1 => MAR= MAR+REG16)+23:0) NEXT
(IF X2 => MAR= MAR+REG16)+23:0) NEXT

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STH;</td>
<td>48 STORE HALFWORD</td>
</tr>
<tr>
<td>LA;</td>
<td>41 LOAD ADDRESS</td>
</tr>
<tr>
<td>STC;</td>
<td>42 STORE CHARACTER</td>
</tr>
<tr>
<td>IC;</td>
<td>43 INSERT CHARACTER</td>
</tr>
<tr>
<td>EX;</td>
<td>44 EXECUTE</td>
</tr>
<tr>
<td>BAL;</td>
<td>45 BRANCH AND LINK</td>
</tr>
<tr>
<td>BCT;</td>
<td>46 BRANCH ON COUNT</td>
</tr>
<tr>
<td>BC;</td>
<td>47 BRANCH ON CONDITION</td>
</tr>
<tr>
<td>LH;</td>
<td>48 LOAD HALFWORD</td>
</tr>
<tr>
<td>CH;</td>
<td>49 COMPARE HALFWORD</td>
</tr>
<tr>
<td>AH;</td>
<td>4A ADD HALFWORD</td>
</tr>
<tr>
<td>SH;</td>
<td>4B SUBTRACT HALFWORD</td>
</tr>
<tr>
<td>MH;</td>
<td>4C MULTIPLY HALFWORD</td>
</tr>
<tr>
<td>OPEK;</td>
<td>4D</td>
</tr>
<tr>
<td>CVD;</td>
<td>4E CONVERT TO DECIMAL</td>
</tr>
<tr>
<td>CVB;</td>
<td>4F CONVERT TO BINARY</td>
</tr>
<tr>
<td>ST;</td>
<td>50 STORE</td>
</tr>
<tr>
<td>OPEK;</td>
<td>51</td>
</tr>
<tr>
<td>OPEK;</td>
<td>52</td>
</tr>
<tr>
<td>OPEK;</td>
<td>53</td>
</tr>
<tr>
<td>N;</td>
<td>54 AND</td>
</tr>
<tr>
<td>CL;</td>
<td>55 COMPARE LOGICAL</td>
</tr>
<tr>
<td>D;</td>
<td>56 OR</td>
</tr>
<tr>
<td>X;</td>
<td>57 EXCLUSIVE OR</td>
</tr>
<tr>
<td>L;</td>
<td>58 LOAD</td>
</tr>
<tr>
<td>C;</td>
<td>59 COMPARE</td>
</tr>
<tr>
<td>A;</td>
<td>5A ADD</td>
</tr>
<tr>
<td>S;</td>
<td>5B SUBTRACT</td>
</tr>
<tr>
<td>M;</td>
<td>5C MULTIPLY</td>
</tr>
<tr>
<td>D;</td>
<td>5D DIVIDE</td>
</tr>
<tr>
<td>AL;</td>
<td>5E ADD LOGICAL</td>
</tr>
<tr>
<td>SL;</td>
<td>5F SUBTRACT LOGICAL</td>
</tr>
<tr>
<td>STD;</td>
<td>6A STORE (LONG)</td>
</tr>
<tr>
<td>OPEK;</td>
<td>61</td>
</tr>
<tr>
<td>OPEK;</td>
<td>62</td>
</tr>
<tr>
<td>OPEK;</td>
<td>63</td>
</tr>
<tr>
<td>OPEK;</td>
<td>64</td>
</tr>
<tr>
<td>OPEK;</td>
<td>65</td>
</tr>
<tr>
<td>OPEK;</td>
<td>66</td>
</tr>
<tr>
<td>OPEK;</td>
<td>67</td>
</tr>
<tr>
<td>LD;</td>
<td>6A LOAD (LONG)</td>
</tr>
<tr>
<td>CD;</td>
<td>6B COMPARE (LONG)</td>
</tr>
<tr>
<td>AD;</td>
<td>6A ADD NORMALIZED (LONG)</td>
</tr>
<tr>
<td>SD;</td>
<td>6B SUBTRACT NORMALIZED (LONG)</td>
</tr>
<tr>
<td>MD;</td>
<td>6C MULTIPLY (LONG)</td>
</tr>
<tr>
<td>DD;</td>
<td>6D DIVIDE (LONG)</td>
</tr>
<tr>
<td>AH;</td>
<td>6E ADD UNNORMALIZED (LONG)</td>
</tr>
<tr>
<td>SW;</td>
<td>6F SUBTRACT UNNORMALIZED (LONG)</td>
</tr>
<tr>
<td>STE;</td>
<td>7A STORE (SHORT)</td>
</tr>
<tr>
<td>OPEK;</td>
<td>71</td>
</tr>
<tr>
<td>OPEK;</td>
<td>72</td>
</tr>
<tr>
<td>OPEK;</td>
<td>73</td>
</tr>
<tr>
<td>OPEK;</td>
<td>74</td>
</tr>
<tr>
<td>OPEK;</td>
<td>75</td>
</tr>
<tr>
<td>OPEK;</td>
<td>76</td>
</tr>
<tr>
<td>OPEK;</td>
<td>77</td>
</tr>
<tr>
<td>LE;</td>
<td>78 LOAD (SHORT)</td>
</tr>
<tr>
<td>CE;</td>
<td>79 COMPARE (SHORT)</td>
</tr>
<tr>
<td>AE;</td>
<td>7A ADD NORMALIZED (SHORT)</td>
</tr>
</tbody>
</table>

B-25
S360 ISP DESCRIPTION

SE; 7B SUBTRACT NORMALIZED (SHORT)
ME; 7C MULTIPLY (SHORT)
OE; 7D DIVIDE (SHORT)
AU; 7E ADD UNNORMALIZED (SHORT)
SU; 7F SUBTRACT UNNORMALIZED (SHORT)

) END OF DECODE

END; END OF RX
S360 ISP DESCRIPTION

RS, SI INSTRUCTIONS

SSM:  SET SYSTEM MASK
BEGIN
PSCK NEXT
ROBYTE NEXT
PSH<0:7>->NBYTE
END;  END OF SSM

LPSW:  LOAD PSW (PRIVILEGED INSTRUCTION)
BEGIN
PSCK NEXT
CKNOWD NEXT READW NEXT
PSH<0:15>-MBR<0:15>; MAR-(MAR+4)<23:0> NEXT
READW NEXT
PSH<34:63>-MBR<2:31>
END;  END OF LPSW

DIAG:  DIAGNOSE (PRIVILEGED INSTRUCTION)

THIS INSTRUCTION DESCRIPTION DOES NOT CORRESPOND
TO ANY PARTICULAR MODEL OF THE S/360 LINE.
IT HAS BEEN MODIFIED FOR USE IN ENDING A
SIMULATION RUN.

BEGIN
PSCK NEXT
T2-ILC; T16-INTCOE NEXT
SCHOUT-PSW NEXT
PSH-MNPSW NEXT
INTCOE-T16; ILC-T2 NEXT
STOPBIT=1;  THIS WILL HALT MACHINE AND SIMULATION
END;  END OF DIAGNOSE

WAD:  WRITE DIRECT (DIRECT CONTROL FEATURE INSTRUCTION)
BEGIN
PSCK NEXT ROBYTE NEXT
SIGOUT-12; IODREG-MBR<24:31>
END;  END OF WAD

RDO:  READ DIRECT (DIRECT CONTROL FEATURE INSTRUCTION)
BEGIN
PSCK NEXT
SIGOUT-12; MBR<24:31>-IODREG NEXT
WRBYTE
END;  END OF RDO

BXH:  BRANCH ON INDEX HIGH
BEGIN
(DECIDE R3<3> =>
\0  T32-REG(R3+1);
\1  T32-REG(R3) ) NEXT
REG(R1)=REG(R1)+REG(R3)<31:0> NEXT
(IF (T32 MINUS REG(R1)<31> =>
BXH1= (PC-MAR)
)
END;  END OF BXH
S380 ISP DESCRIPTION

**BXLE: Branch on Index Less Than or Equal**
BEGIN
(DECODE R3<3> ==
 \0 T32=REG(R3+1);
 \1 T32=REG(R3)
) NEXT
REG(R1)=(REG(R1)+REG(R3))<31;0> NEXT
(IF NOT((T32 MINUS REG(R1))<31 ;8) ==
 BXLE1 = (PC = MAR)
) END; ! END OF BXLE

**SRL: Shift Right Logical**
BEGIN
REG(R1)=REG(R1) 15R0 MAR<18:23>
END; ! END OF SRL

**SLL: Shift Left Logical**
BEGIN
REG(R1)=REG(R1) 15L0 MAR<18:23>
END; ! END OF SLL

**SRA: Shift Right Single Arithmetic**
BEGIN
(DECODE REG(R1)<3>
 \0 REG(R1) - REG(R1) 15R0 MAR<18:23>; ! POSITIVE
 \1 REG(R1) - REG(R1) 15R1 MAR<18:23>; ! NEGATIVE
) NEXT ! END OF DECODE
SETFCC END; ! END OF SRA

**SLA: Shift Left Single Arithmetic**
BEGIN
T6=MAR<18:23> NEXT
SLA1= (IF T6 ==
 (IF REG(R1)<0> NEO REG(R1)<1> == OFW+1) NEXT
 REG(R1)<1:31>=REG(R1)<1:31> 15L0 1);
 T6=(T6 MINUS 1)<5:8> NEXT
SLA1
) NEXT
SETFCC END; ! END OF SLA

**SRDL: Shift Right Double Logical**
BEGIN
(IF R1<3> == INTCD<6>; INTVEC<2>==1 NEXT BAILOUT ICYCLE) NEXT
T64=REG(R1) 15R1=REG(R1+1) NEXT
T64=T64 15R0 MAR<18:23> NEXT
REG(R1)=T64<0:31>;
REG(R1+1)=T64<32:63>
END; ! END OF SRDL
S368 ISP DESCRIPTION

SLDT:=  ! SHIFT LEFT DOUBLE LOGICAL
BEGIN
(IF R1<3> => INTCOE=6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
T64=REG(R1)@REG(R1+1) NEXT
T64=T64 15L0 MAR<18:23> NEXT
REG(R1)=T64<6:31>
REG(R1+1)=T64<32:63>
END; ! END OF SLDT

SRDT:=  ! SHIFT RIGHT DOUBLE ARITHMETIC
BEGIN
(IF R1<3> => INTCOE=6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
(IF MAR<18:23> =>
T64=REG(R1)@REG(R1+1) NEXT
DECIDE T64<0> =>
\0 T64=T64 15RO MAR<18:23>
\1 T64= T64 15RO MAR<18:23>
) NEXT ! END OF DECIDE
REG(R1)=T64<6:31>
REG(R1+1)=T64<32:63>
END; ! END OF SRDT

SLDA:=  ! SHIFT LEFT DOUBLE ARITHMETIC
BEGIN
(IF R1<3> => INTCOE=6; INTVEC<2>=1 NEXT BAILOUT ICYCLE) NEXT
T6-MAR<18:23> NEXT
T64=REG(R1)@REG(R1+1) NEXT
SLDA1:= (IF T6 =>
(IF T6<0> OR T6<1> => OVF=1) NEXT
T64<6:33> OR (T6<6:33> 15L0 1)
T6=(T6 MINUS 1)<5:0> NEXT
SLDA1)
) NEXT ! END OF SLDA1
REG(R1)=T64<6:31>
REG(R1+1)=T64<32:63>
SETFCC
END; ! END OF SLDA
STM:= ! STORE MULTIPLE
BEGIN
T4=R1 NEXT
STM1:= BEGIN
MBR.REG(T4) NEXT
WORD NEXT
(IF T4 NEO R3 =)
T4= (T4+1)<3;8>; MAR= (MAR+4)<23;8> NEXT
STM
END
END; ! END OF STM

TM:= ! TEST UNDER MASK
BEGIN
ROBYTE NEXT
HIBYTE-I2 AND HIBYTE NEXT
TMCC:= BEGIN
CC=0 NEXT
( IF HIBYTE >= )
CC=3 NEXT
(IF 12 XOR HIBYTE >= CC=1)
) ! END OF IF
END; ! END OF TMCC
END; ! END OF TM

MVI:= ! MOVE IMMEDIATE
BEGIN
HIBYTE = 12 NEXT
WORD
END; ! END OF MOVE IMMEDIATE

TS:= ! TEST AND SET
BEGIN
N0P
END; ! END OF TS

NI:= ! AND IMMEDIATE
BEGIN
ROBYTE NEXT
HIBYTE = (HIBYTE AND 12) NEXT
WORD NEXT
NICC:= BEGIN
CC=0 NEXT
( IF HIBYTE <= ) END
END; ! END OF NICC
END; ! END OF NI

CLI:= ! COMPARE LOGICAL IMMEDIATE
BEGIN
ROBYTE NEXT
CLICC:= BEGIN
CC=0 NEXT
( IF HIBYTE LSS 12 <= ) END
( IF HIBYTE GTR 12 <= ) END
END; ! END OF CLICC
END; ! END OF CLI

B-30
S360 ISP DESCRIPTION

01: | OR IMMEDIATE
BEGIN
ROBYTE NEXT
HIBYTE = (HIBYTE OR 12) NEXT
WRBYTE NEXT
OICC:= BEGIN
   CC-0 NEXT
   (IF HIBYTE => CC-1)
END ! END OF OICC
END; ! END OF 01

XI: | EXCLUSIVE OR IMMEDIATE
BEGIN
ROBYTE NEXT
HIBYTE = (HIBYTE XOR 12) NEXT
WRBYTE NEXT
XICC:= BEGIN
   CC-0 NEXT
   (IF HIBYTE => CC-1)
END ! END OF XICC
END; ! END OF XI

LM: | LOAD MULTIPLE
BEGIN
T4=R1 NEXT
LM1:= BEGIN
   READW NEXT
   REG(T4)=MAR NEXT
   (IF T4 NEG R3 =>
   T4=(T4+1)<3:0>; MAR=(MAR+4)<23:0> NEXT
   LM1)
) END OF IF T4
END ! END OF LM1
END; ! END OF LM
I/O INSTRUCTIONS

FORMAT IS AS FOLLOWS:

**OPCODE**
BITS 0:7

**UNUSED**
BITS 8:15

**BASE B1**
BITS 16:19

**DISPLACEMENT D1**
BITS 20:31

THE SUM OF B1 AND D1 HAS THE FOLLOWING FORMAT:

**CHANNEL ADDRESS**
BITS 8:15

**DEVICE AND SUBCHANNEL ADDRESS**
BITS 16:23

(0 IS MULTIPLEXOR)

BITS 24:31

*NOTE: ONLY CHANNELS 8-9 ARE VALID*

**CHWAIT := ! CHANNEL WAIT ROUTINE**
BEGIN
IF NOT CHRLS => CHWAIT
END; ! END OF CHANNEL WAIT

**CHINIT := !**
BEGIN
PSCH NEXT
ADDR NEXT
CHINST<IR<6:7>>=1;
CHSEL-1 NEXT
CHWAIT NEXT
CC-CHANCC; CHSEL-8; CHINST<IR<6:7>>=0
END; ! END OF CHINIT

**SIO := ! START I/O**
BEGIN
CHINIT
END; ! END OF SIO

**TIO := ! TEST I/O**
BEGIN
CHINIT
END; ! END OF TEST I/O

**HIO := ! HALT I/O**
BEGIN
CHINIT
END; ! END OF HALT I/O

**TCH := ! TEST CHANNEL**
BEGIN
CHINIT
END; ! END OF TCH
S360 ISP DESCRIPTION

RS,SI INSTRUCTION DECODE TABLE

RSI1:= BEGIN

MAR + DI NEXT  I EFFECTIVE ADDRESS CALCULATION
(IF B1 = RSI1B1: (MAR + (MAR+REG(B1))<23:B)) NEXT

(DECODE OPCODE<2:7> =)

+ OPCODE DECODING

| SSM; | 88 SET SYSTEM MASK |
| OPEX; | 81 |
| LPSW; | 82 LOAD PSW |
| DIAG; | 83 DIAGNOSE |
| WAD; | 84 WRITE DIRECT |
| ROD; | 85 READ DIRECT |
| BXH; | 86 BRANCH ON INDEX HIGH |
| BXLE; | 87 BRANCH ON INDEX LESS THAN OR EQUAL |
| SSR; | 88 SHIFT RIGHT LOGICAL |
| SSL; | 89 SHIFT LEFT LOGICAL |
| SSR; | 8A SHIFT RIGHT SINGLE ARITHMETIC |
| SLR; | 8B SHIFT LEFT SINGLE ARITHMETIC |
| SRL; | 8C SHIFT RIGHT DOUBLE LOGICAL |
| SLD; | 8D SHIFT LEFT DOUBLE LOGICAL |
| SRA; | 8E SHIFT RIGHT DOUBLE ARITHMETIC |
| SDL; | 8F SHIFT LEFT DOUBLE ARITHMETIC |
| STM; | 90 STORE MULTIPLE |
| TMR; | 91 TEST UNDER MASK |
| MVI; | 92 MOVE IMMEDIATE |
| TS; | 93 TEST AND SET |
| MI; | 94 AND IMMEDIATE |
| CLI; | 95 COMPARE LOGICAL IMMEDIATE |
| CI; | 96 OR IMMEDIATE |
| XI; | 97 EXCLUSIVE OR IMMEDIATE |
| LM; | 98 LOAD MULTIPLE |
| OPEX; | 99 |
| OPEX; | 9A |
| OPEX; | 9B |
| SIO; | 9C START I/O |
| TIO; | 9D TEST I/O |
| MIO; | 9E MALT I/O |
| TCH; | 9F TEST CHANNEL |
| OPEX; | A0 |
| OPEX; | A1 |
| OPEX; | A2 |
| OPEX; | A3 |
| OPEX; | A4 |
| OPEX; | A5 |
| OPEX; | A6 |
| OPEX; | A7 |
| OPEX; | A8 |
| OPEX; | A9 |
| OPEX; | AA |
| OPEX; | AB |
| OPEX; | AC |
| OPEX; | AD |
| OPEX; | AE |
| OPEX; | AF |
| OPEX; | B0 |
| OPEX; | B1 |
| OPEX; | B2 |
| OPEX; | B3 |
| OPEX; | B4 |
| OPEX; | B5 |
| OPEX; | B6 |
| OPEX; | B7 |
| OPEX; | B8 |
| OPEX; | B9 |
| OPEX; | BA |
| OPEX; | BB |
| OPEX; | BC |
| OPEX; | BD |
| OPEX; | BE |
| OPEX; | BF |

) END OF DECODE

R-33
S360 ISP DESCRIPTION

! SS INSTRUCTIONS
! FOR ALL OF THESE INSTRUCTIONS AN ADDRESSING ERROR OR A PROTECTION ERROR
! RESULTS IN TERMINATION OF THE INSTRUCTION. ALL, PART OF, OR NONE OF THE
! RESULT MAY BE STORED. THEREFORE THE RESULTANT DATA IS UNPREDICTABLE AND THE
! SETTING OF THE CONDITION CODE, IF CALLED FOR MAY BE UNPREDICTABLE. IN GENERAL
! THE RESULTS SHOULD NOT BE USED.

MVM:  ! MOVE NUMERICS
BEGIN
LAUX1-8; LAUX2-8 NEXT
MVM1:  BEGIN
ROBYTE2 NEXT ROBYTE NEXT
ROBYTE1; T4-MBR<28:31> NEXT
ROBYTE NEXT
MBR<28:31>-T4 NEXT
WBYTE NEXT
( IF LFLD GTR LAUX2 =>
LAUX1-(LAUX1)+<7:0>; LAUX2-(LAUX2+1)<7:0> NEXT
MVM1
) ! END OF IF LFLD
END ! END OF MVM1
END; ! END OF MVM

MVC:  ! MOVE CHARACTER
BEGIN
LAUX1-8; LAUX2-8 NEXT
MVC1:  BEGIN
ROBYTE2 NEXT ROBYTE NEXT
ROBYTE1 NEXT
WBYTE NEXT
( IF LFLD GTR LAUX2 =>
LAUX1-(LAUX1)+<7:0>; LAUX2-(LAUX2+1)<7:0> NEXT
MVC1
) ! END OF IF LFLD
END; ! END OF MVC

MVZ:  ! MOVE ZONES
BEGIN
LAUX1-8; LAUX2-8 NEXT
MVZ1:  BEGIN
ROBYTE2 NEXT ROBYTE NEXT
ROBYTE1; T4-MBR<24:27> NEXT
ROBYTE NEXT
MBR<24:27>-T4 NEXT
WBYTE NEXT
( IF LFLD GTR LAUX2 =>
LAUX1-(LAUX1)+<7:0>; LAUX2-(LAUX2+1)<7:0> NEXT
MVZ1
) ! END OF IF LFLD
END ! END OF MVZ1
END; ! END OF MOVE ZONES
S368 ISP DESCRIPTION

NC: ! AND CHARACTER
BEGIN
LAUX1=0; LAUX2=0;
NCCC1= (CC-0) NEXT
NC1= BEGIN
ROBYTE NEXT ROBYTE NEXT
ROBYTE NEXT ROBYTE NEXT
HIBYTE-(ROBYTE AND HIBYTE) NEXT WBYTE NEXT
NCCC2= (IF HIBYTE => CC-1) NEXT
(IF LFLOD CTR LAUX2 =>
LAUX1=(LAUX1+1)<7:0>; LAUX2=(LAUX2+1)<7:0> NEXT
NC1
)
END
END; ! END OF NC

CLC: ! COMPARE LOGICAL CHARACTER
BEGIN
LAUX1=0; LAUX2=0;
CLCCC1= (CC-0) NEXT
CLC1= BEGIN
ROBYTE NEXT ROBYTE NEXT
ROBYTE NEXT ROBYTE NEXT
(IF ROBYTE EQ HIBYTE =>
(IF LFLOD CTR LAUX2 =>
LAUX1=(LAUX1+1)<7:0>; LAUX2=(LAUX2+1)<7:0> NEXT
CLC1
)
)
END
CLCCC2= BEGIN
(IF ROBYTE LT HIBYTE => CC-1) NEXT
(IF ROBYTE GT HIBYTE => CC-2)
END; ! END OF CLCCC2
END; ! END OF CLC

OC: ! OR CHARACTER
BEGIN
LAUX1=0; LAUX2=0;
OCCC1= (CC-0) NEXT
OC1= BEGIN
ROBYTE NEXT ROBYTE NEXT
ROBYTE-HIBYTE;
ROBYTE NEXT ROBYTE NEXT
HIBYTE-(ROBYTE OR HIBYTE) NEXT WBYTE NEXT
OCCC2= (IF HIBYTE => CC-1) NEXT
(IF LFLOD CTR LAUX2 =>
LAUX1=(LAUX1+1)<7:0>; LAUX2=(LAUX2+1)<7:0> NEXT
OC1
)
END
END; ! END OF OC
XC:=  ! EXCLUSIVE OR CHARACTER
BEGIN
LAUX1=0; LAUX2=0;
XCCC1= (CC=0) NEXT
XCI:= BEGIN
ADBYT2 NEXT ROBYTE NEXT
LOBYTE=HIBYTE;
ADBYT1 NEXT ROBYTE NEXT
HIBYTE=(LOBYTE XOR HIBYTE) NEXT WBYTE NEXT
XCCC2= (IF HIBYTE => CC=1) NEXT
(IF LFLD GTR LAUX2 =>
LAUK1=(LAUX1+1)<7:0>; LAUK2=(LAUX2+1)<7:0> NEXT
XCI)
END
END; ! END OF XC

TR:=  ! TRANSLATE
BEGIN
LAUX1=0; LAUX2=0 NEXT
TR1:= BEGIN
ADBYT NEXT ROBYTE NEXT
MRR=(AMAR2+HIBYTE)<23:0> NEXT ROBYTE NEXT
ADBYT NEXT WBYTE NEXT
(IF LFLD DIR
LAUX1=(LAUX1+1)<7:0>; LAUK2=(LAUX2+1)<7:0> NEXT
TR1)
END
END; ! END OF TR

TRT:=  ! TRANSLATE AND TEST
BEGIN
LAUX1=0; LAUX2=0;
TRTCC1:= (CC=0) NEXT
TRT1:= BEGIN
ADBYT NEXT ROBYTE NEXT
MRR=(AMAR2+HIBYTE)<23:0> NEXT ROBYTE NEXT
(IF HIBYTE =>
REG1<8:31>=(AMAR1+LAUX1)<23:0>;
REG2<24:31>=HIBYTE NEXT
TRTCC2= BEGIN
CC=1 NEXT
(IF LFLD EQL LAUX1 => CC=2)
END (END OF TRTCC2)
) NEXT
(IF (LFLD GTR LAUX2) AND (HIBYTE EQL 0) =>
LAUK1=(LAUX1+1)<7:0>; LAUK2=(LAUX2+1)<7:0> NEXT
TRT1)
END
END; ! END OF TRT
S300 ISP DESCRIPTION

ZAP:  | ZERO AND ADD (DECIMAL FEATURE INSTRUCTION)
     | BEGIN
     | NOP
     | END;  | END OF ZAP

CP:   | COMPARE DECIMAL (DECIMAL FEATURE INSTRUCTION)
     | BEGIN
     | NOP
     | END;  | END OF COMPARE DECIMAL

AP:   | ADD DECIMAL (DECIMAL FEATURE INSTRUCTION)
     | BEGIN
     | NOP
     | END;  | END OF ADD DECIMAL

SP:   | SUBTRACT DECIMAL (DECIMAL FEATURE INSTRUCTION)
     | BEGIN
     | NOP
     | END;  | END OF SUBTRACT DECIMAL

MP:   | MULTIPLY DECIMAL (DECIMAL FEATURE INSTRUCTION)
     | BEGIN
     | NOP
     | END;  | END OF MULTIPLY DECIMAL

DP:   | DIVIDE DECIMAL (DECIMAL FEATURE INSTRUCTION)
     | BEGIN
     | NOP
     | END;  | END OF DIVIDE DECIMAL
5368 ISP DESCRIPTION

S368 INSTRUCTION DECODE TABLE

BEGIN

AMAR1=DI; AMAR2=D2 NEXT

IF B1 => S3B1 = (AMAR1+AMAR1-REG11)<23:B>>

IF B2 => S3B2 = (AMAR2+AMAR2-REG22)<23:B>> NEXT

(DECODING CODE <2:7> =>)

OPEK1: | C0
OPEK1: | C1
OPEK1: | C2
OPEK1: | C3
OPEK1: | C4
OPEK1: | C5
OPEK1: | C6
OPEK2: | C7
OPEK2: | C8
OPEK2: | C9
OPEK2: | CB
OPEK2: | C0
OPEK2: |CC
OPEK2: | CE
OPEK2: | CF
OPEK3: | D0
MVC; | D1 MOVE NUMERICS
MVZ; | D2 MOVE CHARACTER
NC; | D3 MOVE ZONES
NC; | D4 AND CHARACTER
NC; | D5 COMPARE LOGICAL CHARACTER
NC; | D6 OR CHARACTER
NC; | D7 EXCLUSIVE OR CHARACTER
OPEK3; | D8
OPEK3; | D9
OPEK3; | DB
TR; | DC TRANSLATE
TRT; | DD TRANSLATE AND TEST
ED; | DE EDIT
EDK; | DF EDIT AND MARK
OPEK; | E0
OPEK3; | E1
OPEK3; | E2
OPEK3; | E3
OPEK3; | E4
OPEK3; | E5
OPEK3; | E6
OPEK3; | E7
OPEK4; | E8
OPEK4; | E9
OPEK4; | EA
OPEK4; | EB
OPEK4; | EC
OPEK4; | ED
OPEK4; | EE
OPEK4; | EF
OPEK4; | F0
MVD; | F1 MOVE WITH OFFSET
PACK; | F2 PACK
UNP; | F3 UNPACK
OPEK; | F4
OPEK4; | F5
OPEK4; | F6
OPEK4; | F7
ZAP; | F8 ZERO AND ADD
CP; | F9 COMPARE DECIMAL
AP; | FA ADD DECIMAL
SP; | FB SUBTRACT DECIMAL
MP; | FC MULTIPLY PACKED
DP; | FD DIVIDE PACKED
OPEK; | FE
OPEK; | FF

B-40
S360 ISP DESCRIPTION

END;  END OF SS

END;  END OF DECODE
**S360 ISP DESCRIPTION**

**! INTERRUPT SERVICE ROUTINES**

```plaintext
INT: = BEGIN
T2=ILC NEXT  ! SAVE INSTRUCTION LENGTH

! HANDLE PRIORITY (1) INTERRUPTS
(IF INTVEC<0> AND MKHMKK =>
MKOPSU=PSH NEXT
MKOPSU<16:31>0 NEXT
SCHOUT=PSH NEXT
PSH=MKNPSH;
INTVEC<8:2>=0
) NEXT

! HANDLE PRIORITY (2) INTERRUPTS
(IF INTVEC<1> =>
SVCPSH=PSH NEXT
PSW=SVPNPSH;
INTVEC<1>=0
) NEXT

(IF INTVEC<2> =>
PROP=PSH NEXT
PSW=PRNPSH;
INTVEC<2>=0
) NEXT

! HANDLE PRIORITY (3) INTERRUPTS
(IF INTVEC<3> AND CHMRK =>
INTCDE=EXTREG NEXT
EXOPSU=PSH NEXT
PSW=EXNPSH;
INTVEC<3>=0
) NEXT

! HANDLE PRIORITY (4) INTERRUPTS
(IF INTVEC<4> AND IOMSR =>
INTCDE=DEVREG NEXT
IDOPSU=PSH NEXT
PSW=IONPSH;
INTVEC<4>=0
) NEXT

INTCDE=0; ILC=T2  ! RESET ILC & INTERRUPT CODE
END;  ! END OF INTERRUPT HANDLING
```
S360 ISP DESCRIPTION

1 INSTRUCTION DECODING SECTION

IEXEC:= BEGIN
   DECODE_OPCODE<>1> ->
   RR;
   RX;
   RSSI;
   SS
   END;  ! END OF IEXEC

ICYCLE :=
BEGIN
   IFETCH NEXT
   IEXEC NEXT
   (IF EXRF -> IEXEC NEXT EXRF-0)
   END

ERALCED  ! END OF DECLARATIONS

! MAIN EXECUTABLE PROGRAM

RUN:=
BEGIN
   (IF NOT STOPBIT =>
      (IF NOT WAITST =>
         (ICYCLE
         ) NEXT ! END OF NOT WAITST
         INT NEXT
         RUN
         ) ! END OF NOT STOPBIT
         END ! END OF RUN LOOP
   ) ! END OF S360
INTERDATA 8/32 ISP DESCRIPTION

INTERDATA := ( DECLARE

| USEFUL MACROS --- SYNONYMS |
| MACRO BEGIN :=($ |
| MACRO END := $) |
| MACRO IFF := DECODE(MOT ($ |
| MACRO THEN := ))=>S |
| MACRO ELSE := \ELSE $ |

INTERDATA STORAGE RESOURCES

REGS:0<127:0><31>: !8 SETS OF 16 REGISTERS |
| WARNING: IMPLEMENTATION ASSUMES ONLY 8 REGISTER SETS |

macrop byte:="ffff"

BHMEM:0<maxbyte:0<7>: ; !BYTE-ADDRESSABLE MEMORY |
| BHMEM:7<maxbyte:0<15> := BHMEM:0<maxbyte:0<7>: ; !HALF-WORDS |
| BHMEM:0<maxbyte:0<31> := BHMEM:0<maxbyte:0<15> ; !FULL-WORDS |

PROGRAM STATUS WORD < PSW > AND ITS SUBFIELDS

PSW<0:63> ;

\PSW.SUBFIELDS \COND.CODE
CC<0:3> := PSW<28:31> ; CONDITION CODE |
| C< := PSW<28> ; !CARRY BIT |
| V< := PSW<29> ; !OVERFLOW BIT |
| G< := PSW<30> ; !GREATER-THAN BIT |
| L< := PSW<31> ; !LESS-THAN BIT |

\REGISTER.SET R<0:3> := PSW<28:27> ; CURRENT REGISTER SET |
| LOC<0:15> := PSW<44:63> ; LOCATION COUNTER |

\INTERUPT.MASKBITS \MACRO II := PSW<17><PSW<26> ; IMMEDIATE INTERRUPTS MASK BITS |
| IH< := PSW<17> ; !HIGH-ORDER BIT OF II |
| IL< := PSW<26> ; !LOW-ORDER BIT OF II |
| W< := PSW<15> ; !WAIT-STATE BIT |
| M< := PSW<18> ; !MEMORY MALFUNCTION INTERRUPT MASK BIT |
| A< := PSW<19> ; !ARITHMETIC FAULT INTERRUPT MASK BIT |
| P< := PSW<23> ; !PROTECT MODE INTERRUPT MASK BIT |
| R.P< := PSW<23> ; !MEMORY LOCATION/PROTECTION VIOLATION MASK |
| Q< := PSW<25> ; !QUEUE SERVICE INTERRUPT MASK BIT |

\INSTRUCTION REGISTER <IR> AND ITS SUBFIELDS

IR<0:47> ;

\IR.SUBFIELDS
| OP<8:7> := IR<8:7> ; !OPCODE |
| R1<8:3> := IR<8:11> ; !FIRST OPERAND REGISTER |
| R2<8:3> := IR<12:15> ; !SECOND OPERAND REGISTER |
| N<0:3> := IR<12:15> ; !4-BIT LITERAL |
| X2<8:3> := IR<12:15> ; !SECOND OPERAND INDEX REGISTER |
| R11<0:15> := IR<16:31> ; !16-BIT CONSTANT |
| R12<0:3> := IR<16:47> ; !32-BIT CONSTANT |
| D2<0:3> := IR<16:31> ; !2'S COMPLEMENT DISPLACEMENT |
| FX2<8:3> := IR<12:15> ; !FIRST INDEX REG |
| SX2<8:3> := IR<28:23> ; !2ND INDEX REG |
| A2<8:3> := IR<24:47> ; !USED TO DECODE RK TYPE |

RR FORMAT SUBFIELDS: OP, R1, R2 |
| SF FORMAT SUBFIELDS: OP, R1, N |
| RI FORMAT SUBFIELDS: OP, R1, X2, (R11 OR R1) |
| RX FORMAT SUBFIELDS: OP, R1, (X2, O2) OR (FX1, FX2, R21) |

B-44
INTERDATA 8/32 ISP DESCRIPTION

ISP TEMPORARY REGISTERS

RUN>:;
NOPA;
MACRO NOPA NOPA 88
BYTES:8:2;
INSTR:;
run<:;
fixed.float>
intvec<0:3b>
intlv<0:3b>
devum<0:8>
devstat<>
MACR:8:24;
MACRO wmac:mmac Tar0 2 $;
MACRO hmc:mmac Tar0 1 $;
MACRO MM:0:19>
MACRO MR<0:31>
MACRO wmar:mmar Tar0 2 $;
MACRO hmar:mmar Tar0 1 $;
SET<0:2>:aR1:3>
ER<0:19>
CCOP<0:31>
RIOPO<0:31>
temple;
MACRO align:temp16
  temp<0:3b>
low<0:3b>
temp5<0:4>
temp8<0:7>
TEMP16<0:15>
dat16<0:15>
MACRO max16:dat168
  TEMP17<0:16>
  TEMPP8<0:16>
  TEMP32<0:31>
dat32<0:31>
MACRO div32:dat328
  TEMPP33<0:32>
  TEMPP34<0:63>
MACRO NEWPSH: TEMP200

11-bit temp
14-bit temp reg
14-bit temp reg used by intrekt
15-bit temp reg
16-bit temporary reg
16-bit temporary register (used with list instructions)
132-bit temporary register
132-bit temporary register (used in divide instructions)
133-bit temporary register
164-bit temporary register
164-bit temporary register (used in divide instructions)
168-bit temporary register
168-bit temporary register (used to hold current PSW)
ADDRESS OF NEW PSW

1-10 BYTES CAN BE 1, 2, OR 4 (BYTE, HALF OR FULL WORD)
IFLAG: 0= DATA FETCH; 1= INSTR. FETCH
r.w = 0 --- read; r.w = 1 --- write
fixed.float = 0 --- fixed pt arithmetic; fixed.float = 1 --- floating pt
Interrupt vector (contains a 1 in the bit position corresponding to a pt
contains the current interrupt level being processed
dummy device number
dummy device status
MAC address translation register
MEMORY ADDRESS REGISTER
MEMORY BUFFER REGISTER
REGISTER SET SELECTION; 0 sets only
EFFECTIVE ADDRESS FOR RX & RI FORMAT OPERANDS
COND.CODE OPERAND (32-BIT)
RI FORMAT OPERAND
11-bit temp reg

B-45
INTERDATA 8/32 ISP DESCRIPTION

ISP SUBROUTINES — CALLED BY OTHER ROUTINES

I MAC is called by MEMRD and MEMWT
I performs actions of the Memory Access Controller

MAC:= begin
  INOP
  IMAC
end;

I BNDRYCHK is called by MEMRD and MEMWT
I parameters: nbytes (readonly); mar (read/write)

BNDRYCHK:=begin
  // check address boundary
  // If data read => boundary error causes machine interrupt
  // If instruction read => address are truncated to lowest memory
  // Initially, just truncate all addresses, since INTERDATA
  // has not yet implemented the INT for boundary errors
  IFF (nbytes EQL 2) THEN mar-mar AND "fff0"
  // Halfword accesses must be on a halfword boundary
  ELSE
    (IFF nbytes eq 4 => mar-mar AND "fff0"
      // If fullword read, must be on fullword boundary
      IFF fullword read
      END
    )
  END;

I MEMRD:= begin
  // READ NBYTES FROM MEMORY ADDRESS MAR INTO MBA
  R.W.:= 0 Next
  bndrychk Next
  MACR:=MAR Next
  (IFF R.P => MAC ) NEXT
  // Memory access control
  decode nbytes
  \0 nop;
  \1 mbr:=mem(macr);
  \2 mbr:=mem(macr);
  \3 nop;
  \4 mbr:=mem(macr)
  END;

I MEMWT:= begin
  // MEMORY WRITE ROUTINE
  R.W.:= 1 Next
  bndrychk Next
  MACR:=MAR Next
  (IFF R.P => MAC ) NEXT
  // Memory access control
  decode nbytes
  \0 nop;
  \1 MEMR:=MACR+MBA+24:31;
  \2 MEMW:=MACR+16:31;
  \3 NOP;
  \4 MEME:=MACR+MBA+8:31
  )
  END;

I CCFIXEO:= begin
  // CCFIXEO checks the value of parameter COP
  \0 decode CCOP
  \8 NONNEG
    (IFF (CCOP EQL 0) THEN CC=0 ; CC:=G and L
     ELSE CC=2
     )
  \9 NEG CC=-1
  END;

I SYSTINT:= loads new PSW and saves old PSW in REGISTERS 14 and 15 OF
I the newly selected set
I SYSTINT is used when ANY of the 5 TYPES of INTERRUPTS OCCUR:
I SUPERVISOR CALL, ILLEGAL (OR PROTECTION) INSTRUCTION,
I SYSTEM QUEUE SERVICE, ARITHMETIC FAULT.
INTERDATA 8/32 ISP DESCRIPTION

INPUT PARAMETERS:
NEWPSW - CONTAINS MEMORY ADDRESS OF NEW P5W

SYSINT := BEGIN
OLDPSW-PSW NEXT I SAVE CURRENT P5W
mar-NEWPSW NEXT I ADDR OF NEW P5W
TEMPB<8:31>-NEWINTWARR NEXT I PART OF NEW P5W
MAR-NEWMAR<21:0> NEXT
TEMPB<32:63>-NEWINTWARR NEXT
PSW-TEMPB<4> NEXT
NEW P5W LOADED
PSW-NEWPSW<31:0>-NEWINTWARR + 4<31:0>
1 P5W LOADED WITH DATA IN MEMORY AT ADDRESS NEWPSW
REGIST((1)-OLDPSW<31:0>)
REGIST((1)-OLDPSW<32:63>)
OLD P5W LOADED INTO REGS 14 AND 15 OF NEW SET
END ; SYSINT

QSCHK := BEGIN
mar-<8:0> Next
mar-<8:0>(<mar>-1) Next
<mar>-<mar>+13<31:16> Next
<mar>-<mar>+1<19:0> bytes-2 Next
Newt Newt Next
(IF mar-<>
NEWPSW<8:0> Next
SYSINT Next
REGIST<10:-6><64-66> I i.e., swap psw

IF so, syst. queue interrupt occurs
change processor state (for interrupt handling)
place address of system queue in reg. 13 of new set

) land IF

END ; QSCHK

ARITHCHK called by DIVIDE instructions
(parameters: fixed, float (readonly))

ARITHCHK calls SYSINT

ARITHCHK := Begin
(IF a => NEWPSW<48>;
IF a => NEWPSW<48>;
IF a => NEWPSW<48>;

DECIDE fixed, float =>
\N, fixed c=0;
\F, float c=1

) land IF

End ; ARITHCHK

GETDEVINFO := Begin

Respond to I/O interrupt by getting device no. and status
\dummy parameters returned: devnum & devstat

End ; GETDEVINFO

IIOINT is called by INTCHK when an interrupt of level intlev is to be processed
(parameters: intlev (readonly))

IIOINT calls GETDEVINFO

IIOINT := Begin

IF an I/O interrupt of level intlev exists (and is enabled)
process interrupt!!!

OLDPSW-PSW Next
PSW<8:31> (~28 @ intlev)<31:0> TSL0 & Next
Last new P5W status

REGIST((10<3:0>)= OLDPSW<31:0>;
1 and 2 of the new set

REGIST((10<3:0>)= OLDPSW<32:63> Next
GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next

GETDEVINFO Next
INTERDATA 8/32 ISP DESCRIPTION

INSTRUCTION FORMAT ROUTINES

rrformat := begin
LOC := (MAR +2)<19:8>;  
instr := 0
END ; rrformat

1st instruction format

rfformat :=Begin
rrformat

End; rfformat

rrtype 1 format instruction

rfformat := begin
mar := (mar+2)<19:8>; next  
memrd := next  
loc := (mar+2)<19:8>; next  
update loc,
instr := 0
END ; rfformat

rrtype 2 instruction format

rfformat := begin
mar := (mar+2)<19:8>; next  
nbytes := 2 next  
memrd := next  
loc := (mar+2)<19:8>; next  
update loc,
instr := 0
END ; rfformat

rrtype 3 instruction format

rfformat := begin
mar := (mar+2)<19:8>; next  
nbytes := 2 next  
memrd := next  
temp32 := 0 next  
update loc,
instr := 0
END ; rfformat

INSTRUCTION FORMATS

rrformat := begin
LOC := (MAR +2)<19:8>;  
instr := 0
END ; rrformat

1st instruction format

rfformat :=Begin
rrformat

End; rfformat

rrtype 1 format instruction

rfformat := begin
mar := (mar+2)<19:8>; next  
memrd := next  
loc := (mar+2)<19:8>; next  
update loc,
instr := 0
END ; rfformat

rrtype 2 instruction format

rfformat := begin
mar := (mar+2)<19:8>; next  
nbytes := 2 next  
memrd := next  
loc := (mar+2)<19:8>; next  
update loc,
instr := 0
END ; rfformat

rrtype 3 instruction format

rfformat := begin
mar := (mar+2)<19:8>; next  
nbytes := 2 next  
memrd := next  
temp32 := 0 next  
update loc,
instr := 0
END ; rfformat

B-48
INTERDATA 8/32 ISP DESCRIPTION

(DECODE (fx2 neq 0) & (fx2 eq 0) =>
  \name rx3name=(nep);
  \w
  if rx3!=(nep);
  \w
  if rx3t!=(nep);

  )
  this is for R-M measures only--ignore
END ; } rx3

rx2format\10:= begin
  \x
  temp32=(mar+2+dx)<31:0> NEXT
  (if x2 => rx2aincl=(temp32=(temp32=reg(seta2))<31:0>) ) NEXT
  ea=temp32<12:31>
  128-bit address
END ; } rx2 & dx nonneg

rx2bformat\11:= begin
  \x
  temp32= ((#777777 &x2 ) + mar + 2)<31:0> NEXT
  (if x2 => rx2binc=(temp32=(temp32=reg(seta2))<31:0>) ) NEXT
  ea=(temp32 )<12:31>
  128-bit address
END ; } rx2 and dx neg

) NEXT

END decode r type
loc=(mar+2)<19:0>; instr=d

END } lrxfomat

B-49
ILLINST :=Begin  Illegal instruction encountered
MEMPUSH-"30 NEXT ADDRESS OF NEW PSW FOR ILL. INST. INT. HANDLER
SYSINT  ISWAP PSWS
END ; ILLINST
INTERDATA 8/32 ISP DESCRIPTION

THE LOAD INSTRUCTIONS

LR: = BEGIN
    | LOAD REG
    | RFORMAT NEXT
    | CCOP = REG(SET@R1) NEXT
    | REG(SET@R1) = CCOP NEXT
    | CCFIXED (SET CC)
END; ILR

LIS: = BEGIN
    | LOAD IMMED SHORT
    | SFORMAT NEXT
    | CCOP = N NEXT
    | REG(SET@R1) = CCOP NEXT
    | CCFIXED
END;  ILIS

LCS: = BEGIN
    | LOAD COMPLEMENT SHORT
    | SFORMAT NEXT
    | CCOP = (MINUS(~00000000 & N) <31:0>) NEXT
    | REG(SET@R1) = CCOP NEXT
    | CCFIXED
END;  LCS

LInst: = BEGIN
    | LOAD
    | RXFORMAT NEXT
    | ICA Lculate EA
    | MBYES- 4; MAR=EA NEXT
    | MEMAD NEXT
    | IREAD WORD FROM MEM
    | CCOP=MBR NEXT
    | ILOAD REG(SET@R1) = CCOP NEXT
    | CCFIXED
END;  LINST

LI: = BEGIN
    | LOAD IMMED
    | R2FORMAT NEXT
    | CCOP = RIOPND Next
    | ICOND CODE PARM
    | REG(SET@R1)=CCOP NEXT
    | ILOAD
    | CCFIXED
END;  LI

LH: = BEGIN
    | LOAD HALFWORD
    | RXFORMAT NEXT
    | MAR=EA; MBYES-2 NEXT
    | MEMAD NEXT
    | (DECODE MBR<16>)
    | SIGN BIT
    | \
    | .NONNEG BEGIN
    | CCOP= MBR<16:31> Next
    | REG(SET@R1)=CCOP
    | END; LH-DECODE
    | \1.NEG BEGIN
    | CCOP= (&FFFF @ MBR<16:31>)<31:0> Next
    | REG(SET@R1)=CCOP
    | END \1-DECODE
    | ) NEXT Tend DECODE
    | CCFIXED
END;  LH

LHI: = BEGIN
    | LOAD HALFWORD IMMEDIATE
    | RFORMAT NEXT
    | CCOP=RIOPND Next
    | RIOPND already sign extended
    | REG(SET@R1)=CCOP Next
    | CCFIXED
END;  LHI

LA: = BEGIN
    | LOAD ADDRESS
    | RXFORMAT NEXT
    | REG(SET@R1)=EA 128-BIT ADDRESS
END;  LA

B-51
INTERDATA 8/32 ISP DESCRIPTION

LHL: BEGIN
LOAD HALFWORD LOGICAL
RXFORMAT NEXT
NBYTES=2; MAR EA NEXT
MEMAD NEXT
!READ HALFWORD
CCOP=MBA<16:31> NEXT
ZERO-FILL
REGSET<1> MBA<16:31> NEXT
CCFIXED
END; LHL

LM: BEGIN
LOAD MULTIPLE
RXFORMAT NEXT
NBYTES=4; MAR EA; TEMP4 R1 NEXT
TEMP4 IS USED FOR LOOP COUNTER
LMULT:=
MEMAD NEXT
REGSET<2> MBA<24:31>
IForce HIGH BITS TO ZERO
END; LM

LB: BEGIN
LOAD BYTE
RXFORMAT NEXT
MAR EA; NBYTES=1 NEXT
MEMAD NEXT
REGSET<1> MBA<24:31>
END; LB

BRA: BEGIN
LOAD BYTE REGISTER
RXFORMAT NEXT
REGSET<1> REGSET<2><24:31>
IFORCE HIGHBITS TO ZERO
END; LBRA

STH: BEGIN
STORE HALFWORD
RXFORMAT NEXT
MAR EA; NBYTES=2;
MBA<16:31> REGSET<1> MBA<16:31> NEXT
MEMAD
END; STH

STB: BEGIN
STORE BYTE
RXFORMAT NEXT
MAR EA; NBYTES=1;
MBA<24:31> REGSET<1> MBA<24:31> NEXT
MEMAD
END; STB

STBR: BEGIN
STORE BYTE REGISTER
RXFORMAT NEXT
REGSET<2> MBA<24:31> REGSET<1> MBA<24:31>
END; STBR

STM: BEGIN
STORE MULTIPLE
RXFORMAT NEXT
NBYTES=4; MAR EA; TEMP4 R1 NEXT
SMULT:=
(MBA REGSET<2>) NEXT
MEMAD NEXT
(IF temp4 LSB 15 => temp4 (temp4+1)<3:8> NEXT
LMULT)
END; STM

ST: BEGIN
STORE
RXFORMAT NEXT
MAR EA; NBYTES=4 NEXT
MEMAD
END; ST
INTERDATA 8/32 ISP DESCRIPTION

EXBRs=Begin
  lexhange byte reg
  rformat Next
  reg(set@2)<18:31> reg(set@1)<24:31> @ reg(set@1)<18:23>
End;
  lexbr
INTERDATA 8/32 ISP DESCRIPTION

BOOLEAN INSTRUCTIONS

ORINST=: BEGIN IOR REGISTER
RRFORMAT NEXT
ccop-reg[setreg1] or reg[setreg2] Next
reg[setreg1]=ccop Next
ccfixed
let condition code
end;
lorinst

O:= begin for instr
RRFORMAT Next
 nbytes=4; mar=ea Next !prepare to fetch opnd
memrd Next !read a word
ccop= reg[setreg1] or mbr<0:31> Next
ccfixed
end; O

O1:= begin for immediate
RRFORMAT Next
ccop-reg[setreg1] or riopnd Next
reg[setreg1]= ccop Next
ccfixed
end; O1

OH:=begin for halfword
RRFORMAT Next
mar=ea; nbytes=2 Next !prepare to fetch data
memrd Next !fetch halfword data
(IFF (mbr<16> EQL 0) THEN temp32=mbr<16:31> !test sign bit
ELSE temp32= "ffff @ mbr<16:31> !propagate sign bit(negative)
) Next !end IFF
ccop = reg[setreg1] OR temp32 Next
reg[setreg1]=ccop Next
ccfixed
END; O

OH1:=Begin for halfword immediate
RRFORMAT Next
ccop-reg[setreg1] OR riopnd Next !riopnd already sign extended
reg[setreg1]=ccop Next
ccfixed
END; O

X:=Begin !exclusive or
RRFORMAT Next
mar=ea; nbytes=4 Next !prepare to fetch data
memrd Next !fetch fullword data
ccop-reg[setreg1] XOR mbr Next
reg[setreg1]=ccop Next
ccfixed
END; O

XR:=Begin !exclusive or register
RRFORMAT Next
ccop-reg[setreg1] XOR reg[setreg2] Next
reg[setreg1]=ccop Next
ccfixed
END; O

XI:=Begin !exclusive or immediate
RRFORMAT Next
ccop-reg[setreg1] XOR riopnd Next
reg[setreg1]=ccop Next
ccfixed
END; O
INH; BEGIN
exclusive or halfword

rxformat Next

r=caseROYTE=2 Next

prepare to fetch data

r=word Next

fetch fullword data

(1FF (mbr<18> = EQL 0) THEN temp32=mbr<16:31>

Test sign bit

ELSE temp32="FFFF $ mbr<16:31>

Sign extend data

) NEXT [end IFF]

cmp=register(1) XOR mbr Next

register(1)=ccop Next

cf

END;

INH; BEGIN
exclusive or halfword immediate

rxformat Next

cmp=register(1) XOR riopw Next

sign already extend

register(1)=ccop Next

cf

END;

INH; BEGIN

AND

RXFORMAT NEXT

MARK-ER; NBYTES-4;

MARK NEXT ! Fetch OPERAND

CCOP=REGISTER(1) AND MBR NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND REG

RXFORMAT NEXT

CCOP=REGISTER(1) AND REGISTER(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND register(2) NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;

INH; BEGIN

AND IMMEO

RXFORMAT NEXT

CCOP=REGISTER(1) AND RIOPNO NEXT

REGISTER(1)=CCOP NEXT

CCFIXED

END;
INTERDATA 8/32 ISP DESCRIPTION

SHIFT INSTRUCTIONS

SLLs: Begin
  shift left logical
  r|iformat Next
  temp33<32:0> = reg1<32:0> TSL 0 (riopnd<27:31>) Next
  luse low-order 5-bits of riopnd to determine shift count
  produce a 33-bit result which includes the carry in bit 0
  ccop=temp33<1:32>
  register1=ccop Next 132-bit result
  ccfixecl Next
  c= temp33<32> last carry bit of condition code
END: li 1

SLLS: Begin
  shift left logical short
  r|iformat Next
  temp33<32:0> = reg1<32:0> TSL 0 n Next
  ccop=temp33<1:32>
  register1=ccop Next 132-bit result
  ccfixecl Next
  c= temp33<32> last carry bit of condition code
END; li 1

SLHL: Begin
  shift left halfword logical
  r|iformat Next
  temp33<16:31> = reg1<16:31> TSL 0 (riopnd<28:31>) Next
  luse low-order 4-bits of riopnd to determine shift count
  shift the low-order 16-bits only; retain the last bit shifted
  ccop=8:15= temp17<1:16>
  pass ccfixed a 16-bit result
  register1<16:31>=ccop<8:15> Next 110-bit result
  ccfixecl Next
  c= temp17<32> last carry bit of condition code
END; li 1

SLHLS: Begin
  shift left halfword logical short
  r|iformat Next
  temp33<16:31> = reg1<16:31> TSL 0 n Next
  lshift the low-order 16-bits only; retain the last bit shifted
  ccop<8:15>= temp17<1:16>
  pass ccfixed a 16-bit result
  register1<16:31>=ccop<8:15> Next 110-bit result
  ccfixecl Next
  c= temp17<32> last carry bit of condition code
END; li 1

SRLs: Begin
  shift right logical
  r|iformat Next
  temp33<31:0> = register1 Next (left-justity in 33-bit reg
  lin order to retain the last bit shifted out
  temp33= temp33 TSL 0 (riopnd<27:31>) Next
  luse low-order 5-bits of riopnd to determine shift count
  produce a 33-bit result which includes the carry in bit 32
  ccop=temp33<3:31>
  register1=ccop Next 132-bit result
  ccfixecl Next
  c= temp33<32> last carry bit of condition code
END; li 1

SRLS: Begin
  shift right logical short
  r|iformat Next
  temp33<31:0> = register1 Next (left-justity into 33-bit reg
  lin order to retain the last bit shifted out
  temp33= temp33 TSL 0 n Next
  ccop=temp33<3:31>
  register1=ccop Next 132-bit result
  ccfixecl Next
  c= temp33<32> last carry bit of condition code
END; li 1

SRHL: Begin
  shift right halfword logical

B-56
INTERDATA 8/32 ISP DESCRIPTION

rilformat Next

\begin{verbatim}
   |result format
   temp17<0:15><[reg]<16:31> Next  left-justify in 17-bit reg
   \quad in order to retain the last bit shifted out
   \quad \text{temp17} \text{ TSO} (\text{riopnd}<28:31>) Next
   \quad \text{use low-order 4-bits of riopnd to determine shift count}
   \quad \text{produce a 17-bit result which includes the carry in bit 18}
   \quad \text{ccop}<8:15>=\text{temp17}<8:15>;
   \quad \text{[reg]<16:31}+\text{ccop}<8:15> Next  \text{16-bit result}
   \quad \text{ccfixed Next}
   \text{c}=\text{temp17}<16> \quad \text{last carry bit of condition code}
\end{verbatim}

\text{END; lsh}

SRHLS:=Begin
\quad \text{shift right halfword logical short}
\quad \text{rilformat Next}
\begin{verbatim}
   temp17<0:15><[reg]<16:31> Next  left-justify into 17-bit reg
   \quad \text{in order to retain the last bit shifted out}
   \quad \text{temp17} \text{ TSO} n Next
   \quad \text{ccop}<8:15>=\text{temp17}<8:15>;
   \quad \text{[reg]<16:31}+\text{ccop}<8:15> Next  \text{16-bit result}
   \quad \text{ccfixed Next}
   \text{c}=\text{temp17}<16> \quad \text{last carry bit of condition code}
\end{verbatim}

\text{END; lsh}

RLL:=Begin
\quad \text{rotate left logical}
\quad \text{rilformat Next}
\begin{verbatim}
   \text{ccop}=[\text{reg}<11:16> \text{RI} \text{riopnd}<27:31>] Next  \text{132-bits}
   \quad \text{rotate left by the amount specified in the low 5 bits of riopnd}
   \quad \text{reg<16:31>+ccop} \text{Next} \text{result}
   \quad \text{ccfixed}
   \text{c}=\text{[reg]<11:16>}
\end{verbatim}

\text{END; lrl}

RRL:=Begin
\quad \text{rotate right logical}
\quad \text{rilformat Next}
\begin{verbatim}
   \text{ccop}=[\text{reg}<11:16> \text{RR} \text{riopnd}<27:31>] Next
   \quad \text{rotate right by the amount specified in the low 5 bits of riopnd}
   \quad \text{reg<16:31>+ccop} \text{Next}
   \quad \text{ccfixed}
   \text{c}=\text{[reg]<11:16>}
\end{verbatim}

\text{END; lrr}

SLA:=Begin
\quad \text{shift left arithmetic}
\quad \text{rilformat Next}
\begin{verbatim}
   \text{sign}=[\text{reg}<1:8> \text{<16:32} \text{RI} \text{reg<31:1>} \text{Next}
   \quad \text{leave sign of } \text{reg<1:8>;} \text{prepare to shift remaining bits}
   \quad \text{temp32}=(\text{temp32} \text{ TSO} \text{riopnd}<27:31>)<31:8> \text{Next}
   \quad \text{shift by amount specified by low 5 bits of riopnd}
   \quad \text{ccop}+\text{sign} \text{<temp32<1:31>} \text{Next} \text{the sign bit remains unchanged}
   \quad \text{reg<16:31>+ccop} \text{Next}
   \quad \text{ccfixed} \text{Next} \text{c}=\text{[temp32<2:0>}} \quad \text{last carry bit of cc}
   \text{End; lsl}
\end{verbatim}

SLHA:=Begin
\quad \text{shift left halfword arithmetic}
\quad \text{rilformat Next}
\begin{verbatim}
   \text{sign}=[\text{reg}<1:8> \text{<16:32} \text{RI} \text{reg<31:1>} \text{Next}
   \quad \text{leave sign of } \text{reg<1:8>;} \text{prepare to shift remaining bits}
   \quad \text{temp16}=\text{(temp16} \text{ TSO} \text{riopnd}<28:31>)<15:0> \text{Next}
   \quad \text{use low-order 4-bits of riopnd to determine shift count}
   \quad \text{ccop}<8:15>+\text{sign} \text{<temp16<1:15>} \text{Next}
   \quad \text{reg<16:31>+ccop<8:15>} \text{Next}
   \quad \text{ccfixed} \text{Next}
   \text{c}=\text{[temp16<0>}} \quad \text{last carry bit of cc}
\end{verbatim}

\text{END; lsla}

SRR:=Begin
\quad \text{shift right arithmetic}
\quad \text{rilformat Next}
\begin{verbatim}
   \text{sign}=[\text{reg}<1:8> \text{<16:32} \text{RI} \text{reg<31:1>} \text{Next}
   \quad \text{leave sign}
   \quad \text{temp32<0:36}=\text{[reg]<1:31> \text{Next} \text{do right 31-bit shift}}
   \quad \text{DECOD sign} \quad \text{sign determines fill bit}
\end{verbatim}

\text{R-57}


```
INTERDATA 8/32 ISP DESCRIPTION

\6.pos temp32 temp32 TSR9 riopnd<27:31>
\1.neg temp32 temp32 TSR1 riopnd<27:31>
) Next (end decode
\isigned right, sign fill; carry in bit 31 of temp32
\ccop= sign @ temp32<0:38> Next \isigned @ 31-bit result
\reg(set[1])=ccop Next
\ctfixed Next
c=+temp32<31>
\iset carry bit of cc

END; \isa

SRHR:=Begin
\isigned right halfword reg
\rformat Next
temp16<0:16>=reg[set[1]<17:31]>; \isigned shift to be done
\sign=reg[set[1]<0> Next \isave sign
\ DECODE sign => \isigned determines fill bit
\6.pos temp16= temp16 TSR9 riopnd<28:31>
\1.neg temp16 temp16 TSR1 riopnd<28:31>
) Next
\isigned right by amt specified in low 4 bits of riopnd
\when shifting, propagate sign, and save carry in bit 15 of temp16
\ccop<15> = sign @ temp16<0:14> Next
\reg(set[1]<16:31>=ccop<0:15> Next
\ctfixed Next
c=+temp16<15>
\iset carry in cc

END; \isa

TS:=Begin
\istest and set
\rformat Next
\mbr=ea; nbytea=2 Next \isprepare to fetch halfword data
\mem= Next \ifetch halfword
\mbr16> Next \ifuse most significant bit to set cc
\NOTE: \icould have done ccop<0:15>=mbr<16:31>, and called \ctfixed
\mbr=mbr OR "0000 Next \iset most significant bit of halfword
\mem= Next \ife set most significant bit of halfword
\mbr= Next \wwrite back halfword

END; \isa

TLATE:=Begin
\itranslate
\rformat Next
\ea=ea + (\{'0&reg[set[1]<24:31>\}<8:0> TSL0 1))<19:0>;nbytea=2 Next
\ifuse character in ea to index into table at address ea
\mem= Next \iread halfword table entry into low half of mbr
\IFF (mbr<16> EQL 1) THEN \icall test most significant bit of table entry
\table contains a new translated character
\ELSE \iclear reg[set[1]<24:31>=mbr<24:31>
\branch to translation routine
\specified by table entry
\ENDIF

END; \italic
```
INTERDATA 8/32 ISP DESCRIPTION

ICOMPARISON INSTRUCTIONS

Cin:=Begin          Icompare
  rformat Next      temp33- (register1) + (NOT mbr) + 1)<32:8> Next
  mar-as; nbytes=4 Next      i prepare to fetch data
  memrd Next       (reg. b=-2) (reg. a=-2) has no positive value, given n bits)
  temp33- (register1) + (NOT mbr) + 1)<32:8> Next
  i subtract operands and compare result with zero
  temp33-> Next      i perform "2+n = a - b" to cover all possible values of a & b
  i subtract operands and compare difference with zero
  cccompute 32-bit result
  ccfixed Next
  (IF I => c-1)      i carry set in cc when relation is <
  cinat
  END; Ccr

C1:=Begin          Icompare reg.
  r12format Next    temp33- (register1) + (NOT reg[seter2]) + 1)<32:8> Next
  i perform "2+n = a - b" to cover all possible values of a & b
  cccompute 32-bit result
  ccfixed Next
  (IF I => c-1)      i carry set in cc when relation is <
  c1c
  END; C1c

C1:=Begin          Icompare immediate
  r12format Next    temp33- (register1) + (NOT reg[seter2]) + 1)<32:8> Next
  i compute difference of operands and compare with zero
  c= NOT temp33<8>
  ccfixed Next
  c< NOT (temp33<8>)      i carry set for < relations
  END; C1c

CH:=Begin          Icompare halfword
  r2format Next     \
  mar-as; nbytes=2 Next      i prepare to fetch halfword
  memrd Next       i sign-extend and convert data to fullword operand and do comparison
  get halfword data
  (DECODE mbr<16>> =>
    \0.pos temp33- (register1) + (NOT "FFFFmbr<16:31>") + 1)<32:8> ;
    \1.neg temp33- (register1) + (NOT "0000 @ mbr<16:31>") + 1)<32:8>
    ) Next
  iend decode
  cccompute 32-bit result
  ccfixed Next
  c= NOT (temp33<8>)      i carry set when relation is <
  END; C1c

CH1:=Begin         Icompare halfword immediate
  r12format Next    temp33- (register1) + (NOT riopnd) + 1)<32:8> Next
  i riopnd already sign extended
  cccompute 32-bit result
  ccfixed Next
  c= NOT temp33<8>      i set carry when relation is <
  END; C1c

CL:=Begin          Icompare logical
  r2format Next     \
  mar-as; nbytes=4 Next      i prepare to fetch data word
  memrd Next
  (DECODE register1)c<8> @ mbr<8> =>
    i signs of operands
    \00 (IF I => c-1); i both positive; relation same as arithmetic compare
    \01 c-1; i relation is <
    \10 nop; i relation is >
    \11 (IF I => c-1) reverse relation
B-59
INTERDATA 8/32 ISP DESCRIPTION

END; lend decode

CLR:=Begin
|compare logical reg

|format Next
|temp33<regset>1> + (NOT regset2) + 1)<32:0> Next
ccope=temp33<1:32> Next
cclipired Next |compute arithmetic cc
(DECODE regset1<0> & regset2<0> => signs of operands

\00 (IF I => c-1); both positive; relation same as arithmetic compare
\01 c-1; relation is <
\10 nop; relation is >
\11 (IF g => c-1) reverse relation )

lend decode

END; lclr

CLI:=Begin
|compare logical immediat

|format Next
|temp33<regset1> + (NOT riopnd) + 1)<32:0> Next
ccope=temp33<1:32> Next
cclipired Next |compute arithmetic cc
(DECODE regset1<0> & riopnd<0> => signs of operands

\00 (IF I => c-1); both positive; relation same as arithmetic compare
\01 c-1; relation is <
\10 nop; relation is >
\11 (IF g => c-1) reverse relation )

lend decode

END; lcli

CLH:=Begin
|compare logical halfword

|format Next
|prepare to fetch data halfword
|read Next

( IFF mbr<16> THEN temp32<16:31>
ELSE temp32+ "if i & mbr<16:31>
)
next halfword data sign extended
|temp33<regset1> + (NOT temp32) + 1)<32:0> Next
ccope=temp33<1:32> Next
cclipired Next |compute arithmetic cc
(DECODE regset1<0> & temp32<0> => signs of operands

\00 (IF I => c-1); both positive; relation same as arithmetic compare
\01 c-1; relation is <
\10 nop; relation is >
\11 (IF g => c-1) reverse relation )

lend decode

END; lclih

CLHI:=Begin
|compare logical halfword immediat

|format Next
|riopnd already sign extended
|temp33<regset1> + (NOT riopnd) + 1)<32:0> Next
ccope=temp33<1:32> Next
cclipired Next |compute arithmetic cc
(DECODE regset1<0> & riopnd<0> => signs of operands

\00 (IF I => c-1); both positive; relation same as arithmetic compare
\01 c-1; relation is <
\10 nop; relation is >
\11 (IF g => c-1) reverse relation )

lend decode

END; lclih

CLB:=Begin
|compare logical byte

|format Next
|prepare to fetch byte data
|read Next

temp8<regset1<24:31> + (NOT mbr<24:31> + 1)<7:0> Next
ccope=8:7>temp8 Next
cclipired Next |compute arithmetic cc
(DECODE regset1<24> & mbr<24> => signs of data

B-60
INTERDATA 8/32 ISP DESCRIPTION

\00 (IF I => c+1) ; both positive; relation same as arithmetic compare
\01 c-1 ; relation is <
\10 nop ; relation is >
\11 (IF g => c+1) ; reverse relation

END

CHVR: Begin

\convert to halfword value reg

\reformat Next

\temp32<16:31> \= \reg{set[e]2}<16:31> Next
<DIGITCODE temp32<16> \= \"halfword sign bit"
\\.pos \temp32<8:15> \= \#0;
\\.neg \temp32<8:15> \= \"ffff"
) Next \sign extended

\temp<1> \\leave current carry in cc

\ccop=\temp32 \+\reg{set[e]1}=\temp32 Next

\ccfixed Next

\c=\temp<1> \\restore old carry

\DIGITCODE temp32<8> \= \sign of result
\\.pos \(IF \reg{set[e]2}<8:15> \\NEQ \#0 \then \#-1\)
\\.neg \(IF \reg{set[e]2}<8:15> \\NEQ \"ffff \then \#-1\)
) \overflow occurred --- not a valid halfword

END

B-61
INTERDATA 8/32 ISP DESCRIPTION

BIT INSTRUCTIONS

TBT:=Begin
  test bit
  rformat Next
  mar-(ea + register) TSR 3<19:8> nbytes=1 Next
  value in reg is a bit displacement into the array
  starting at address ea.
  memrd Next
  fitch byte from array
  (DECODE register)1<29:31> => get indicated bit

  \0 (temp1=mbr<24>) Next
  mbr<24> NOT temp1;
  \1 (temp1=mbr<25>) Next
  mbr<25> NOT temp1;
  \2 (temp1=mbr<26>) Next
  mbr<26> NOT temp1;
  \3 (temp1=mbr<27>) Next
  mbr<27> NOT temp1;
  \4 (temp1=mbr<28>) Next
  mbr<28> NOT temp1;
  \5 (temp1=mbr<29>) Next
  mbr<29> NOT temp1;
  \6 (temp1=mbr<30>) Next
  mbr<30> NOT temp1;
  \7 (temp1=mbr<31>) Next
)
  Next end decode bit select
  ccop-temp1 Next
  !set cc's g if bit is 1
cfixed
END;
tb

CBT:=Begin
  complement(flip) bit
  rformat Next
  mar-(ea + register) TSR 3<19:8>;
  nbytes=1 Next
  fitch byte within bit array
  memrd Next
  (DECODE register)1<29:31> => select bit within selected byte

  \0 (tempi-mbr<24>) Next
  mbr<24> NOT tempi;
  \1 (tempi-mbr<25>) Next
  mbr<25> NOT tempi;
  \2 (tempi-mbr<26>) Next
  mbr<26> NOT tempi;
  \3 (tempi-mbr<27>) Next
  mbr<27> NOT tempi;
  \4 (tempi-mbr<28>) Next
  mbr<28> NOT tempi;
  \5 (tempi-mbr<29>) Next
  mbr<29> NOT tempi;
  \6 (tempi-mbr<30>) Next
  mbr<30> NOT tempi;
  \7 (tempi-mbr<31>) Next
)
  Next end decode bit select
  memwr Next
  write back byte with bit flipped
  ccop-temp1 Next
  !set cc's g if bit was 1
cfixed
END;
tcb

SBT:=Begin
  set bit
  rformat Next
  mar-(ea + register) TSR 3<19:8>;
  nbytes=1 Next
  fitch byte within bit array
  memrd Next
  (DECODE register)1<29:31> => select bit within selected byte

  \0 (temp1s-mbr<24>) Next
  mbr<24> 1
  \1 (temp1s-mbr<25>) Next
  mbr<25> 1
  \2 (temp1s-mbr<26>) Next
  mbr<26> 1
  \3 (temp1s-mbr<27>) Next
  mbr<27> 1
  \4 (temp1s-mbr<28>) Next
  mbr<28> 1
  \5 (temp1s-mbr<29>) Next
  mbr<29> 1
  \6 (temp1s-mbr<30>) Next
  mbr<30> 1
  \7 (temp1s-mbr<31>) Next
Q.CD

END;
ctb
INTERDATA 8/32 ISP DESCRIPTION

mbr<31>= 1 )
) Next lend decode bit select
memt Next
write back byte with bit flipped
ccop-templ Next
ccfixed
set cc's g if bit was 1
END;

!slbt

RBT:=Begin
(reset(clear) bit)

rxformat Next
mar-(ea + registerj) TSR0 3)<19:0>;
nbytes-1 Next
fetch byte within bit array
Memrd Next

ccop-templ Next
ccfixed
set cc's g if bit was 1
END;

END;

RBT:=Begin
(reset(clear) bit)

rxformat Next
mar-(ea + registerj) TSR0 3)<19:0>;
nbytes-1 Next
fetch byte within bit array
Memrd Next

ccop-templ Next
ccfixed
set cc's g if bit was 1
END;

END;

B-63
INTERDATA 8/32 ISP DESCRIPTION

ARITHMETIC INSTRUCTIONS

AInstr: Begin
  IADD
  rXformat Next
  mar-ea; nbytes=4 Next
  memrd Next
  (fetch the operand (fullword)
  temp32 = reg[set@1] + mbr Next 133-bit result
  ccop = temp33<1:32> Next 132-bit result
  ccfixed Next
  test cond code
  (if temp33<0> => c<1>) Next
  test carry
  (if (mbr<8> EQL reg[set@1]<8>) AND (temp33<8> NEQ temp33<1>)
  => v=1
  ) Next
  test for overflow
  reg[set@1] = ccop 132-bit result
END; !AInstr

AR: Begin
  IADD
  rXformat Next
  temp33 = reg[set@1] + reg[set@2] Next 133-bit result
  ccop = temp33<1:32> Next 132-bit result
  ccfixed Next
  test initial cc
  (if temp33<0> => c<1>) Next
  test for carry
  (if (reg[set@1]<8> EQL reg[set@2]<8>) AND (temp33<8> NEQ temp33<1>)
  => v=1
  ) Next
  reg[set@1] = ccop 132-bit result
END; !AR

AI: Begin
  IADD
  rX2format Next
  temp33 = reg[set@1] + riopnd Next 133-bit result
  ccop = temp33<1:32> Next 132-bit result
  ccfixed Next
  test initial cc
  (if temp33<0> => c<1>) Next
  test for carry
  (if (reg[set@1]<8> EQL riopnd<8>) AND (temp33<8> NEQ temp33<1>)
  => v=1
  ) Next
  reg[set@1] = ccop 132-bit result
END; !AI

AIS: Begin
  IADD
  rXformat Next
  temp32-n Next
  temp33 = reg[set@1] + temp32 Next 133-bit result
  ccop = temp33<1:32> Next 132-bit result
  ccfixed Next
  test initial cc
  (if temp33<0> => c<1>) Next
  test for carry
  (if (reg[set@1]<8> EQL temp32<8>) AND (temp33<8> NEQ temp33<1>)
  => v=1
  ) Next
  reg[set@1] = ccop 132-bit result
END; !AIS

AH: Begin
  LHADD
  rXformat Next
  mar-ea; nbytes=-2 Next
  memrd Next
  (if mbr<16> => mbr<0:15>=""f"f"") Next
  align extend halfword data
  temp33 = reg[set@1] + mbr Next 133-bit result
  ccop = temp33<1:32> Next 132-bit result
  ccfixed Next
  test initial cc
  (if temp33<0> => c<1>) Next
  test for carry
  (if (reg[set@1]<8> EQL mbr<8>) AND (temp33<8> NEQ temp33<1>)
  => v=1
  ) Next
  reg[set@1] = ccop 132-bit result
END; !AH
INTERDATA 8/32 ISP DESCRIPTION

AH1=Begin
rformat Next
ladd halfword immediate

trigend Next
ladd halfword immediate

ccop - temp33<1:32> Next
l32-bit result

ccfixed Next
lset cond code

(( temp33<8 >> c-1 ) Next
ltest carry

(( register11<0> EQL trigend<0>) AND (temp33<8> NEQ temp33<1>)
v=1

) Next
ltest for overflow

register11 = ccop
l132-bit result

END ; [AH1]

AH1=Begin
rformat Next
ladd to memory

mar-as; nbytes+4 Next
lprepare to fetch halfword data

meand Next
lprepare to fetch halfword data

(temp17- (register11 + mbr) Next
l16-bit result

ccop<0:15> = temp17<1:16> Next
lset initial cc

ccfixed Next
lset cond code

(( temp33<8 >> c-1 ) Next
ltest borrow

(( mbr<8> EQL register11<0>) AND (temp33<8> NEQ temp33<1>)
v=1

) Next
ltest for overflow

mbr-ccop Next
lprepare to store result back in memory

mewat
lwrite fullword result at address ea

END ; [AH1]

AH1=Begin
rformat Next
ladd halfword memory

mar-as; nbytes+2 Next
lprepare to fetch halfword data

meand Next
lprepare to fetch halfword data

(( mbr<16> == mbr<15><"fff") Next
l17-bit result

ccop<0:15> = temp17<1:16> Next
l16-bit result

ccfixed Next
lset initial cc

(( temp33<8 >> c-1 ) Next
ltest borrow

(( mbr<8> EQL register11<0>) AND (temp33<8> NEQ temp33<1>)
v=1

) Next
ltest for overflow based on halfword result

mbr-temp17<1:16> Next
lwrite halfword result back to memory at address ea

END ; [AH1]

S=Begin
rformat Next
lsubtract

mar-as; nbytes+4 Next
lprepare to fetch halfword data

meand Next
lprepare to fetch halfword data

(temp33- (register11 + (NOT mbr) + 1)<32:8> Next
l133-bit result

ccop = temp33<1:32> Next
l132-bit result

ccfixed Next
lset cond code

(( temp33<8 >> c-1 ) Next
ltest borrow

(( mbr<8> EQL register11<0>) AND (temp33<8> NEQ temp33<1>)
v=1

) Next
ltest for overflow

register11 = ccop
l132-bit result

END ; [S]

SR=Begin
rformat Next
lsubstract register

(temp33- (register11 + (NOT register2)) + 1)<32:8> Next
l133-bit result

ccop = temp33<1:32> Next
l132-bit result

ccfixed Next
lset cond code

(( temp33<8 >> c-1 ) Next
ltest for borrow

(( register11<0> EQL register21<0>) AND (temp33<8> NEQ temp33<1>)
v=1

) Next
ltest for overflow

register11 = ccop
l132-bit result

END; [SR]
INTERORTR 8/32 ISP DESCRIPTION

SI: Begin
  r12form Next
  temp33 = (register1) + (NOT riopnd) + 1)32:8) Next 133-bit result
  ccop = temp33:132) Next 132-bit result
  ccfix Next
  (IF temp33:8 <= c-1) Next
  (IF (register1:1 < 132:0) AND (temp33:8 <= NOE temp33:1))
  ) Next => v=1
  test for borrow
  test for overflow
  if the signs of the two operands are the same
  and these differ from that of the result
  then overflow occurred
  reg[register1] + ccop

END; lsi

SIS: Begin
  sformat Next
  temp32 = (register1) + (NOT temp32) + 1)32:8) Next 133-bit result
  ccop = temp33:132) Next 132-bit result
  ccfix Next
  (IF temp33:8 <= c-1) Next
  (IF (register1:1 < 132:0) AND (temp33:8 <= NOE temp33:1))
  ) Next => v=1
  test for borrow
  test for overflow
  if the signs of the two operands are the same
  and these differ from that of the result
  then overflow occurred
  reg[register1] + ccop

END; lsi

SH: Begin
  r12form Next
  temp32 = (register1) + (NOT temp32) + 1)32:8) Next 133-bit result
  ccop = temp33:132) Next 132-bit result
  ccfix Next
  (IF temp33:8 <= c-1) Next
  (IF (register1:1 < 132:0) AND (temp33:8 <= NOE temp33:1))
  ) Next => v=1
  test for borrow
  test for overflow
  if the signs of the two operands are the same
  and these differ from that of the result
  then overflow occurred
  reg[register1] + ccop

END; lsh

SHI: Begin
  r12form Next
  temp33 = (register1) + (NOT riopnd) + 1)32:8) Next 133-bit result
  ccop = temp33:132) Next 132-bit result
  ccfix Next
  (IF temp33:8 <= c-1) Next
  (IF (register1:1 < 132:0) AND (temp33:8 <= NOE temp33:1))
  ) Next => v=1
  test for borrow
  test for overflow
  if the signs of the two operands are the same
  and these differ from that of the result
  then overflow occurred
  reg[register1] + ccop

END; lshi

Minst: Begin
  r12form Next
  instruction requires an even/odd register pair
  (IF ri<3) => rop) Next
  (IF rl not even =>error==garbage results!!
  mar<eac; rhytes+2 Next
  memrd Next
  (IF mbr<16 => mbr<8:15>="ff") Next
  sign extend halfword data
  temp33 = (register1) + (NOT mbr) + 1)32:8) Next 133-bit result
  ccop = temp33:132) Next 132-bit result
  ccfix Next
  (IF temp33:8 <= c-1) Next
  (IF (register1:1 < 132:0) AND (temp33:8 <= NOE temp33:1))
  ) Next => v=1
  test for borrow
  test for overflow
  if the signs of the two operands are the same
  and these differ from that of the result
  then overflow occurred
  reg[register1] + ccop

END; lshi

B-72
INTERRUPT 8/32 ISP DESCRIPTION 11-4

End: Mn

MHr=Begin

Multiply register instruction

rformat Next

Instruction requires an even/odd register pair
r1 must specify an EVEN register = r1 and (r1 +1) selected
(If r1<32 = mop) Next If r1 is not even = error -- garbage results!!!
Sign=0 temp32=reg(seter1) Next need to determine sign of result
(temp32<8 = dat32=MNUS dat32)<31:8; sign=(sign<1)> Next
(temp32<8 = temp32=MNUS temp32)<31:8; sign=(sign<1)> Next
Next compute 64-bit product
If sign = temp32=MNUS temp32<32:63> Next lcheck sign of result
reg(seter1)<temp64<8:31>; place sign most significant part of result in even register of the
Pair place least significant part of result in odd register
End: Mn

MHr=Begin

Multiply halfword instruction

rformat Next

mar=*; nbytes=2 Next prepare to fetch halfword data operand
meord Next fetch halfword data
Sign=0 temp16=reg(seter1)<16:31> Next a logical multiply requires positive operands
(temp16<8 = sign=(sign<1)>0; temp16=MNUS temp16<15:B>) Next
(temp16<8 = sign=(sign<1>)<B>; temp16=MNUS temp16<15:B>) Next
(temp16<8 = temp16=MNUS temp16<16:31>) Next compute 32-bit product
If sign = temp32=MNUS temp32<31:8>) Next lcheck sign of result
reg(seter1)<temp32 Next store result in reg
End: Mn

MHR:Begin

Multiply halfword register

rformat Next

Sign=0 temp16=reg(seter1)<16:31> logical multiply requires positive operands
(temp16<8 = sign=(sign<1>)0; temp16=MNUS temp16<15:B>) Next
(temp16<8 = sign=(sign<1>)<B>; temp16=MNUS temp16<15:B>) Next
(temp16<8 = temp16=MNUS temp16<16:31>) Next compute 32-bit product
If sign = temp32=MNUS temp32<31:8>) Next Icheck sign of result
reg(seter1)<temp32 Next store result in reg
End: MNR

Dinats=Begin

Divide

rformat Next

an even/odd register pair is required
error r1 must be EVEN -- garbage results
fixed_float=0 Next Fixed pt arithmetic operation
fixed float is a parameter passed to the arithmetic routine (if an
mar=*; nbytes=4 Next arithmetic) Fetch 32-bit divisor
meord Next
(IF (mbbr EQ 0) THEN arithchkt; cannot divide by zero
ELSE Begin
(temp64=reg(seter1 OR 1)) Next 64-bit dividend: most significant bits in even reg
sign=0 Next initialize sign of quotient
(temp64<8 = sign=(sign<1>)<B>; temp64=MNUS temp64<15:8>) Next determine sign of quotient and force operands to be positive
(temp64<8 = temp64=MNUS temp64<31:8>) Next ldivide is required for logical divide
(temp64<8 = temp64=MNUS temp64<31:8>) Next (idivisor must be positive
(temp64<8 = temp64=MNUS temp64<31:8>) Next for logical divide
(temp64<8 = temp64=MNUS temp64<31:8>) Next iquotient is positive
(temp64<8 = temp64=MNUS temp64<31:8>) Next ELSE Begin
(temp32<8 = mbr=MNUS mbr<31:8>) Next calculate quotient and remainder to the
(temp64<8 = temp64=MNUS temp64<31:8>) Next result is positive
(temp64<8 = temp64=MNUS temp64<31:8>) Next (remainder
temp64<8 = temp64=MNUS temp64<31:8>) Next iconvert quotient and remainder to the
(temp64<8 = temp64=MNUS temp64<31:8>) Next result is positive
(temp64<8 = temp64=MNUS temp64<31:8>) Next

1. neg Begin

Result is negative

B-73
INTERDATA 8/32 ISP DESCRIPTION

register1<32:31> = (MINUS temp32)<31:0>
register1<1 OR 11> = (MINUS q64<32:63>)<31:0>
end

End;  !end DECODE
)  !ovf ELSE
)  !end ovfchk
End  !ELSE
)  !end IFF
End  !lnst

ELSE BEGIN

@ = Begin

format Next

(FF s1x3 = n0p) Next
fixed float 8 Next

fixed 8 float is a parameter passed to the arithchk routine (iff an

ELSE BEGIN

temp64 = register1 @ register1<1 OR 11> Next

164-bit dividend: most significant bits in even reg

sign = Next

initialize sign of quotient

sign<0>

Next

!determine sign of quotient and force operands to be positive

Next

164-bit dividend: most significant bits in even reg

sign<0>

Next

logical dividend

!must be positive for logical divide

ELSE BEGIN

End  !ovf ELSE

End  !ELSE

End  !IFF

End;  !lnst

DR = Begin

!divide register

an even/odd register pair is required

error r1 must be EVEN—garbage results

fixed pt arithmetic operation

End  !ovf ELSE

End! ELSE

End;  !lnst

B-74
INTERDATA 8/32 ISP DESCRIPTION

BRANCH INSTRUCTIONS

TII=BEGIN  ITTEST IMMEDIATE
R12FORMAT NEXT
CCOP=RECISEM@1) AND RIOPND NEXT
CCFIXED
END; ITII

TIM=BEGIN  ITTEST HALFWORD IMMED
R12FORMAT NEXT
CCOP=RECISEM@1) AND RIOPND NEXT
CCFIXED
END; ITIM

BFC=:BEGIN  IBRANCH ON FALSE COND
R12FORMAT NEXT
(IF (CC AND R1) EOL 0 =>
LOC=ER AND "FFFFFFE"
) MUST BE ON HALFWORD BOUNDARY
END; IBFC

BFCR=:BEGIN  IBRANCH ON FALSE REC
R12FORMAT NEXT
(IF (CC AND R1) EOL 0 =>
LOC=(REGISTEHM@1) AND "FFFFFFE")<19:8>
) MUST BE ON HALFWORD BOUNDARY
END; IBFCR

BFBS=:BEGIN  IBRANCH FALSE BACKWARDS SHORT
SFFORMAT NEXT
(IF (CC AND R1) EOL 0 <=
LOC=LOC - ((NOB)+2))<19:8>
) BRANCH N HALFWORDS BACKWARDS
END; IBFBS

BFFS=:BEGIN  IBRANCH FALSE FORWARD SHORT
SFFORMAT NEXT
(IF (CC AND R1) EOL 0 <=
LOC=(LOC+((NOB)-2))<19:8>
) BRANCH N HALFWORDS FORWARD
END; IBFFS

BTC=:BEGIN  IBRANCH ON TRUE COND
R12_FORMAT NEXT
(IF (CC AND R1) EOL 0 =>
LOC=ER AND "FFFFFFE"
) IF ANY COND TRUE => BRANCH
) END IF MUST BE ON HALFWORD BOUNDARY
END; IBTC

BTCR=:BEGIN  IBRANCH ON TRUE COND REG
R12_FORMAT NEXT
(IF (CC AND R1) EOL 0 =>
LOC=(REGISTEHM@1) AND "FFFFFFE")<19:8>
) MUST BE ON HALFWORD BOUNDARY
END; IBTCR

BTBS=:BEGIN  IBRANCH ON TRUE BACKWARD SHORT
SFFORMAT NEXT
(IF (CC AND R1) EOL 0 =>
LOC=(LOC - ((NOB)+2))<19:8>
) BRANCH N HALFWORDS BACKWARDS
END; IBTBS

BFFS=:BEGIN  IBRANCH TRUE FORWARD SHORT
SFFORMAT NEXT
(IF (CC AND R1) EOL 0 =>
LOC=(LOC+((NOB)-2))<19:8>
) BRANCH N HALFWORDS FORWARD
END; IBFSS
INTERRUPT 8/32 ISP DESCRIPTION

loc = ea AND "ffff" address must be on halfword bdry
END; 'BRLR

BRLR := Begin
  branch and link register
  r format: Next
  register1= loc Next leave current loc (next instr addr)
  loc = (register2) AND "ffff fe"<19:8> branch
  address must be on halfword bdry
END; 'BRLR

BXLE := Begin
  branch on index low or equal
  r format: Next
  register1= (register1 + register((r1+1)<3:8>)<31:8> Next
  (if register1) leq register((r1+2)<3:8>) =>
  loc-ea AND "ffff" warning: branch addr must be on halfword bdry
END; 'BXLE

BXHi := Begin
  branch on index high
  r format: Next
  register1= (register1 + register((r1+1)<3:8>)<31:8> Next
  (if register1) gtr register((r1+2)<3:8>) =>
  warning: branch addr must be on halfword bdry
END; 'BXHi
**INTERORTA 8/32 ISP DESCRIPTION**

**ICIRCULAR LIST INSTRUCTIONS**

**RTL**: Begin

1. **Format Next**
   - Add to top of list
   - Get address of list head
   - Prepare to fetch list head
   - Read first word of list head
   - First halfword contains the max. no. of slots in list
   - Second halfword contains the no. of slots used
   
   **(IFF** (max16 LEQ mbr<16:13>) THEN cc--4;
   - List full = overflow
   **ELSE** Begin
   
   **mar-- (ea + 2)<19:8>**; **mbr-- mbr<16:13> + 1**; **nbytes--2 Next**
   - Increment the no. of slots used
   **memw Next**
   - Land update list head
   **mar-- (ea + 4)<19:8> Next**
   - Prepare to read another halfword of the list head
   **memrd Next**
   - Get slot no. of current list top
   **(IFF** (mbr EQL 0) THEN mbr--max16;
   - Max. no. of slots in list
   **ELSE** mbr--mbr - 1 ) Next
   
   **memw Next**
   - Mbr now contains slot no. of current list top
   - Update ptr to current top of list in head
   **mar-- (ea + 6 + (mbr TSH 2))<19:8> Next**
   - Mbr now contains the address of the current top of the list
   **memrd Next**
   - Mbr contains slot no. of current list top
   **memw Next**
   - Add element to list
   **cc--4**
   - List updated successfully
   
   **End**

   **else**

   **End ; lat**

**RBL**: Begin

1. **Format Next**
   - Add to bottom of list
   - Get address of list head
   - Prepare to fetch list head
   - Read first word of list head
   - First halfword contains the max. no. of slots in list
   - Second halfword contains the no. of slots used
   
   **(IFF** (max16 LEQ mbr<16:13>) THEN cc--4;
   - List full = overflow
   **ELSE** Begin
   
   **mbr-- (ea + 2)<19:8>**; **mbr-- mbr<16:13> + 1**; **nbytes--2 Next**
   - Increment the no. of slots used
   **memw Next**
   - Land update list head
   **mar-- (ea + 6)<19:8> Next**
   - Prepare to read another halfword of the list head
   **memrd Next**
   - Get slot no. of next list bottom
   **temp16--mbr<16:13> Next**
   - Save slot no. of next list bottom
   **mar-- (ea + 8 + (mbr TSH 2))<19:8> Next**
   - Mbr now contains the address of the next bottom of the list
   **memw Next**
   - Mbr now contains slot no. of next list bottom
   **memr Next**
   - Add element to list
   **cc--8**
   - List updated successfully
   
   **End**

   **else**

   **End ; tab**

**RTL**: Begin

1. **Format Next**
   - Remove from top of list
   - Prepare to read list head
   - Read list word of list head
   - List halfword is the max. no. of slots in the list
   - List for underflow (i.e. list empty)
   
   **(IFF** (mbr<16:13> EQL 0) THEN cc--4;
   **ELSE** Begin
   
   **mbr--mbr<16:13> - 1**; **nbytes--2 ;**
   **mar-- (ea + 2)<19:8> Next**
   **memw Next**
   **(IFF** mbr THEN cc+=2; **ELSE** cc+=4) Next
   **mar-- (ea + 2)<19:8> Next**

**END**
INTERORIP 8/32 ISP DESCRIPTION

<table>
<thead>
<tr>
<th>Fetch slot no. of current top of list</th>
<th>Fetch current top of list slot no.</th>
<th>Calculate address of slot at top of list</th>
<th>Remove data from slot</th>
<th>Increment slot no. of current top of list</th>
<th>Test for list &quot;wrap-around&quot;</th>
<th>Update current top of list in list header</th>
</tr>
</thead>
<tbody>
<tr>
<td>nword Next</td>
<td>nword Next</td>
<td>nword Next</td>
<td>nword Next</td>
<td>nword Next</td>
<td>nword Next</td>
<td>nword Next</td>
</tr>
<tr>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
<td>mar-(ea + 8 + ((&quot;0temp16&quot;) TSL 2))&lt;19:0&gt;</td>
</tr>
</tbody>
</table>

RBL: Begin

Begin

End;
INTERDATA 8/32 ISP DESCRIPTION

PRIVILEGED INSTRUCTIONS

LPSW:=BEGIN:
"Load program status word"
(1FF P THEN ILLINST ; I PROTECT MODE ON
ELSE BEGIN
RC5:6;R6:4 NEXT ; COMPLETE INSTR FETCH AND CALCULATE EA
(IF ER<17:19> NEQ 0-> MUST BE ON DOUBLE WORD Boundary
ER<17:19->0 ) NEXT ; FORCE IT
MAR-ER NEXT
TEMP64<0:31>-WMEM(WMAR) NEXT
MAR-=(MAR<44><19:8> NEXT
TEMP64<32:63>-WMEM(WMAR) NEXT
PSW=TEMP64 NEXT
IPSW=WENEM(ER)+WMEM(ER<8>)
QSCHK ; CHECK Q SERVICE
END ; ELSE
) IFF END
END ; LPSW

EPSR:=BEGIN:
"Exchange program status register"
(1FF P THEN ILLINST ; I PROTECT MODE ON
ELSE BEGIN
reformat NEXT
REG1=E5 ashamed MOVPS<0:31> NEXT
PSW<0:31>-REG1=WREG<2R> NEXT
Qschk ; CHECK Q SERVICE
END ; ELSE
) IFF END
END ; EPSSR

SVC:=BEGIN:
"Supervisor call"
RC5:6;R6:4 NEXT ; SAVE CURRENT PSW
MAR="98 NEXT ; IADDR OF NEW STATUS
PSW<31>-WMEM(WMAR) NEXT ; I NEW STATUS
MAR=+9MAR)+TSB<1+;SC NEXT ; IADDR OF NEW LOC
LOC=WMEM(WMAR) NEXT ; I NEW LOC
REG1=WREG<2R>; ; IPASS PARAMETER
REG1=+OLDPSW<31>; ; IPASS OLD PSW
REG1=F+OLDPSW<32:63>
END ; SVC

LPSWR:=BEGIN:
"Load program status word reg"
(1FF P THEN ILLINST ; I PROTECT MODE ON
ELSE BEGIN
Qschk ; CHECK Q SERVICE
END ; ELSE
) IFF END
END ; LPSWR
INTERRUPT 8/32 ISP DESCRIPTION

UNIMPLEMENTED INSTRUCTIONS

DHR := Begin
  riformat
End; ldnhr

LER := Begin
  riformat
End; ler

CER := Begin
  riformat
End; lcer

PER := Begin
  riformat
End; lper

SEA := Begin
  riformat
End; lsea

MEA := Begin
  riformat
End; lmea

OEM := Begin
  riformat
End; loem

FXR := Begin
  riformat
End; lfxr

FLR := Begin
  riformat
End; lflr

EXHR := Begin
  riformat
End; lexhr

OH := Begin
  riformat
End; loh

STE := Begin
  riformat
End; lste

LE := Begin
  riformat
End; lle

CE := Begin
  riformat
End; lce

AE := Begin
  riformat
End; lae

SE := Begin
  riformat
End; lse

ME := Begin
  riformat
End; lme

DE := Begin
  riformat
End; lde
INTERDATA 8/32 ISP DESCRIPTION

STNE: Begin
  rxfomat
End; stne
LME: Begin
  rxformat
End; lme
CRC12: Begin
  rxformat
End; lorc12
CRC16: Begin
  rxformat
End; lorc16
WBR: Begin
  rxfomat
End; lwb
RBR: Begin
  rxformat
End; lrb
WHR: Begin
  rxformat
End; lwhr
RHR: Begin
  rxformat
End; lhr
WO: Begin
  rxformat
End; lud
RO: Begin
  rxformat
End; lrd
WB: Begin
  rxfomat
End; lub
RB: Begin
  rxformat
End; lrb
WH: Begin
  rxformat
End; lwh
RH: Begin
  rxformat
End; lrh
WO: Begin
  rxformat
End; lud
RD: Begin
INTERDATA 8/32 ISP DESCRIPTION

SS:=Begin
rxformat
End; 1rd

QC:=Begin
rxformat
End; loc

SINT:=Begin
rllformat
End; lrint

SCP:=Begin
rxformat
End; lscp

LOR:=Begin
rrformat
End; lldr

COR:=Begin
rrformat
End; lcdr

ADR:=Begin
rrformat
End; ladr

SDR:=begin
rrformat
End; lsdrr

MDR:=Begin
rrformat
End; lmdrr

DDR:=begin
rrformat
End; lddrr

FXDR:=Begin
rrformat
End; lfxdr

FLDR:=Begin
rrformat
End; lfldr

LRR:=Begin
rxformat
End;

STD:=begin
rxformat
End;

LD:=Begin
rxformat
End;

cd:=begin
rxformat
end;
ad:=begin
rxformat
end;
sd:=begin
INTERDATA 8/32 ISP DESCRIPTION

```plaintext
  rxformat
end;
md:=begin
  rxformat
end;
dd:=begin
  rxformat
end;
std:=begin
  rxformat
end;
lmd:=begin
  rxformat
end;
```
INTERDATA 8/32 ISP DESCRIPTION

SIMULATION ROUTINES

IFETCH
BEGIN
READ 1ST HALFWORD OF INSTRUCTION
MAR- LOC;
NBYTES = 2;  NO. OF BYTES TO BE READ
INSTR = 1 NEXT THIS MEMORY ACCESS IS DURING IFETCH
MEMORY NEXT READ NBYTES FROM MEMORY
IR=IR+15> - MAR<16:31> NEXT NEXT INSTRUCTION REGISTER
CCOP-0 INITIALIZE THE COND. CODE PARAMETER
END IFETCH

IXQT
BEGIN
DECODE OPCODE AND HANDLE INSTRUCTION
(DECODE OP =)
80 ILLINST;
81 BALR;
82 BTR;
83 BFCR;
84 NR;
85 CLR;
86 OR INST;
87 XR;
88 LA;
89 CR;
8A AR;
8B SR;
8C MHR;
8D DHR;
8E ILLINST;
8F ILLINST;
10 SRLS;
11 SLLS;
12 CHV;
13 ILLINST;
14 ILLINST;
15 ILLINST;
16 ILLINST;
17 ILLINST;
18 ILLINST;
19 LPSHR;
1A ILLINST;
1B ILLINST;
1C MR;
1D DR;
1E ILLINST;
1F ILLINST;
20 BTBS;
21 BTFS;
22 BFBS;
23 BFBS;
24 LIS;
25 LCS;
26 AIS;
27 SIS;
28 LIR;
29 CSR;
2A AER;
2B SER;
2C MER;
2D DER;
2E FSR;
2F FIR;
30 ILLINST;
31 ILLINST;
32 ILLINST;
33 ILLINST;
34 ENHR.
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>36</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>37</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>38</td>
<td>Idr</td>
</tr>
<tr>
<td>39</td>
<td>cdr</td>
</tr>
<tr>
<td>3a</td>
<td>adr</td>
</tr>
<tr>
<td>3b</td>
<td>sdr</td>
</tr>
<tr>
<td>3c</td>
<td>mdr</td>
</tr>
<tr>
<td>3d</td>
<td>ddr</td>
</tr>
<tr>
<td>3e</td>
<td>fxdr</td>
</tr>
<tr>
<td>3f</td>
<td>tdr</td>
</tr>
<tr>
<td>40</td>
<td>STH</td>
</tr>
<tr>
<td>41</td>
<td>BAL</td>
</tr>
<tr>
<td>42</td>
<td>BTC</td>
</tr>
<tr>
<td>43</td>
<td>BFC</td>
</tr>
<tr>
<td>44</td>
<td>NN</td>
</tr>
<tr>
<td>45</td>
<td>CLH</td>
</tr>
<tr>
<td>46</td>
<td>OH</td>
</tr>
<tr>
<td>47</td>
<td>XM</td>
</tr>
<tr>
<td>48</td>
<td>LH</td>
</tr>
<tr>
<td>49</td>
<td>CH</td>
</tr>
<tr>
<td>4A</td>
<td>AH</td>
</tr>
<tr>
<td>4B</td>
<td>SH</td>
</tr>
<tr>
<td>4C</td>
<td>RH</td>
</tr>
<tr>
<td>4D</td>
<td>DH</td>
</tr>
<tr>
<td>4E</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>4F</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>50</td>
<td>ST</td>
</tr>
<tr>
<td>51</td>
<td>AM4</td>
</tr>
<tr>
<td>52</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>53</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>54</td>
<td>Minst</td>
</tr>
<tr>
<td>55</td>
<td>CL</td>
</tr>
<tr>
<td>56</td>
<td>O</td>
</tr>
<tr>
<td>57</td>
<td>X</td>
</tr>
<tr>
<td>58</td>
<td>Linst</td>
</tr>
<tr>
<td>59</td>
<td>Cinst</td>
</tr>
<tr>
<td>5A</td>
<td>Rinst</td>
</tr>
<tr>
<td>5B</td>
<td>S</td>
</tr>
<tr>
<td>5C</td>
<td>Minst</td>
</tr>
<tr>
<td>5D</td>
<td>Dinst</td>
</tr>
<tr>
<td>5E</td>
<td>crc12</td>
</tr>
<tr>
<td>5F</td>
<td>crc16</td>
</tr>
<tr>
<td>60</td>
<td>ste</td>
</tr>
<tr>
<td>61</td>
<td>ahm</td>
</tr>
<tr>
<td>62</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>63</td>
<td>lrc</td>
</tr>
<tr>
<td>64</td>
<td>ATL</td>
</tr>
<tr>
<td>65</td>
<td>ABL</td>
</tr>
<tr>
<td>66</td>
<td>RTL</td>
</tr>
<tr>
<td>67</td>
<td>ABL</td>
</tr>
<tr>
<td>68</td>
<td>le</td>
</tr>
<tr>
<td>69</td>
<td>ce</td>
</tr>
<tr>
<td>6A</td>
<td>ae</td>
</tr>
<tr>
<td>6B</td>
<td>se</td>
</tr>
<tr>
<td>6C</td>
<td>me</td>
</tr>
<tr>
<td>6D</td>
<td>de</td>
</tr>
<tr>
<td>6E</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>6F</td>
<td>ILLINST1</td>
</tr>
<tr>
<td>70</td>
<td>std</td>
</tr>
<tr>
<td>71</td>
<td>stme</td>
</tr>
<tr>
<td>72</td>
<td>lme</td>
</tr>
<tr>
<td>73</td>
<td>LHL</td>
</tr>
<tr>
<td>74</td>
<td>TBT</td>
</tr>
<tr>
<td>75</td>
<td>SBT</td>
</tr>
<tr>
<td>76</td>
<td>RBT</td>
</tr>
</tbody>
</table>

B-85
INTERDATA 8/32 ISP DESCRIPTION

\77 CBT;
\78 1d;
\79 cd;
\7a ad;
\7b sd;
\7c md;
\7d dd;
\7e smd;
\7f lmd;

\80 ILLINST;
\81 ILLINST;
\82 ILLINST;
\83 ILLINST;
\84 ILLINST;
\85 ILLINST;
\86 ILLINST;
\87 ILLINST;
\88 ILLINST;
\89 ILLINST;
\8a ILLINST;
\8b ILLINST;
\8c ILLINST;
\8d ILLINST;
\8e ILLINST;
\8f ILLINST;

\90 SRHL$;
\91 SLHLS;
\92 STBR;
\93 LBR;
\94 EXBR;
\95 EPSR;
\96 wbr;
\97 rbr;
\98 wbr;
\99 rbr;
\9a wdr;
\9b rdr;
\9c ILLINST;
\9d ser;
\9e ocr;
\9f ILLINST;

\80 ILLINST;
\81 ILLINST;
\82 ILLINST;
\83 ILLINST;
\84 ILLINST;
\85 ILLINST;
\86 ILLINST;
\87 ILLINST;
\88 ILLINST;
\89 ILLINST;
\8a ILLINST;
\8b ILLINST;
\8c ILLINST;
\8d ILLINST;
\8e ILLINST;
\8f ILLINST;

\80 ILLINST;
\81 ILLINST;
\82 ILLINST;
\83 ILLINST;
\84 ILLINST;
\85 ILLINST;
\86 ILLINST;
\87 ILLINST;
\88 ILLINST;
\89 ILLINST;
\8a ILLINST;
\8b ILLINST;
\8c ILLINST;
\8d ILLINST;
\8e ILLINST;
\8f ILLINST;

B-86
INTERDATA 8/32 ISP DESCRIPTION

\89 ILLINST;
\8A ILLINST;
\8B ILLINST;
\8C ILLINST;
\8D ILLINST;
\8E ILLINST;
\8F ILLINST;

\90 BXH;
\91 BXLE;
\92 LPSH;
\93 THI;
\94 NHI;
\95 CLHI;
\96 OHI;
\97 XHI;
\98 LHI;
\99 CHI;
\9A AHI;
\9B SHI;
\9C SRHL;
\9D SLR;
\9E SRA;
\9F SRA;

\98 STM;
\99 LM;
\9B SIB;
\9B LB;
\9C CLB;
\9D AB;
\9E WB;
\9F RFB;
\90 WH;
\90 RH;
\92 WD;
\93 RD;
\94 ILLINST;
\95 MSS;
\96 OCC;
\97 ILLINST;

\98 TS;
\9A SVC;
\9B MINT;
\9C SCP;
\9D ILLINST;
\9E ILLINST;
\9F LAR;
\9F TLATE;
\9B ILLINST;
\9C ILLINST;
\9D RRL;
\9E RLL;
\9F SRL;
\9B SRL;
\9C SRA;
\9D SRA;

\9E ILLINST;
\9F ILLINST;
\9F ILLINST;
\9B T;
\94 NI;
\95 CLI;
\96 CI;
\97 XI;
\98 LI;
\99 CI;
\9A NI;
`INTERDATA 8/32 ISP DESCRIPTION`

```
\FB  SI;
\FC ILLINST;
\FD ILLINST;
\FE ILLINST;
\FF ILLINST;
}
\end opode decode
\END; \IXQT

INTCHK:=BEGIN
中断检查和处理

IF intvec =>
(DEC0DE i2 =>
\$8 bailout emulate;
\$1 Begin
(\IFF set G00 3 THEN low-2;
ELSE low<(set - 1)<2:8>
) Next
(\IFF set G0L 8 => low=8)
\End;
\$0 low=3;
\$1 low=set
) Next \end DECODE

intlev=0; temp4=1 Next
getllev=\IFF (intlev LEQ low) AND (temp4 REX 0) =>
\IFF (intlev XOR temp4) THEN (intvec=extvec XOR temp4 Next low=8);
\Is there an interrupt pending at level intvec?
\If so, clear the interrupt and process it
ELSE Begin
\IFF (temp4 TSL0 1)<3:8>
intlev=(intlev + 1)<1:8> Next
\end getllev
\End
ELSE
\end getllev
\End IF
\End
```

ERALCED
INTERDATA 8/32 ISP DESCRIPTION

! MAIN PROGRAM FOLLOWS:

EMULATE =BEGIN IEMUULATION CYCLE
( IF RUN =>
  ( IF NOT m => lis the wait bit on in the ppm
  IF so, just wait for int
  IFETCH NEXT IFETCH 1ST HALFWORD OF INSTRUCTION
  INQIT NEXT INQITExecutive INSTRUCTION
  INTCHE NEXT INTCHE AND HANDLE INTERRUPTS
  EMULATE IFETCH NEXT INSTRUCTION
  ) ) end IF

END IEMUULATE
)

! END INTERDATA
PDP-11 ISP DESCRIPTION 1-1

pdp11 :=
DECLARE

Please report errors to Dan Siewiorek, CMU, (412)-621-2688 x177

The PDP-11 ISP has several features to aid in reading and data gathering. These include:

1.) A word memory defined on top of a byte memory. Thus byte accesses can be separately counted from word accesses.

2.) All memory accessing is done through two routines, READ and WRITE on page 6-1. Memory management protection is enforced by these routines.

3.) All effective address calculation is done by two routines called SOURCE (for the source operand) and DEST (for the destination operand). The routines are described on page 7-1.

4.) All parameters are passed to procedures in the register called TEMP.

5.) Instruction mnemonics are used as labels for the ISP sequence that simulates the instruction's effects. This provides easy reader reference as well as a counter for statistics gathering. Further, each mnemonic is followed by its value in the current decode statement (eg DIV).

6.) Several types of labels were added both for statistics gathering and for control over the counters. For example, it was decided that condition code setting was a combinatorial action and that a register transfer should not be charged. Thus labels were added to condition code setting routines so that they could be copped. The PDP-11 architecture represented a good example why a simulator can do things in data collection that hard analysis would find difficult if not impossible to do. The R count of PDP-11 instructions was a function of the addressing mode used. Thus variables were added that counted the number of times source or destination addressing mode zero (register) was used with each instruction. Below is a key to the significance of the labels. <instruction> stands for the instruction mnemonic.

<table>
<thead>
<tr>
<th>Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Instruction</em></td>
<td>ISP to simulate instruction, used to count the number of times instruction was executed.</td>
</tr>
<tr>
<td><em>Instruction</em></td>
<td>Multiple instructions defined by this procedure, such as MOV (move) and MOVB (move byte). Individual instruction labels internal to procedure.</td>
</tr>
<tr>
<td><em>Condition code setting portion of instruction</em></td>
<td></td>
</tr>
<tr>
<td><em>Carry condition code</em></td>
<td></td>
</tr>
<tr>
<td><em>Negative condition code</em></td>
<td></td>
</tr>
<tr>
<td><em>Zero condition code</em></td>
<td></td>
</tr>
<tr>
<td><em>Overflow condition code</em></td>
<td></td>
</tr>
<tr>
<td><em>destination address mode = 0 (Register)</em></td>
<td></td>
</tr>
</tbody>
</table>

B-90
The following is a page by page description of the ISP:

Page 2-1. The primary memory and mappings (note word/byte memory and I/O page), central processor registers, and the floating point processor status register.

Page 3-1. The PDP-11/40 memory management registers and error registers that allow an instruction retry.

Page 4-1. Temporary registers not seen by programmer. These registers are necessary to completely define the algorithms performed by the hardware (such as address calculation) but these registers are not part of the architecture.

Page 5-1. Instruction decoding formats.

Page 6-1. Start of the procedures. Memory accessing procedures.

Page 7-1. Effective address calculating procedures.

Page 8-1. Condition code setting procedures.

Pages 9 through 15. These are the actual instruction definitions. Similar instructions are grouped together into classes that follow the several levels of decoding that the hardware must go through. Since procedures must be defined before use, the decoding sequence is in reverse order from that encountered in the ISP. As a guide to the reader, the following is a summary of the decoding order. Items in <> represent instructions, all others are just procedure names for further decoding.

**notill**

exec (main decode call execute)
  resrop (reserved op code class)
    <double operand instructions>
    extop (extended instruction set)
  resrop
    branop (branch instructions)
    classop (secondary decode into classes)
  extop
    inttext (integer extended instructions)
    ftext (floating point instructions)
  branop
    regop (register operations)
    <branch instructions>
  regop
    cpucon (cpu control instructions)
    procon (program control instructions)
  classop
    subset (subroutine/emulator traps)
    singop (single operand instructions)
    shifrop (shift instructions)

B-91
It should be pointed out that in the floating point processor the precision mode bit (single or double) determines whether the operation is performed with 32 or 64 bits. Thus even though the ISP reads as if there are two floating point instruction sets, the same bit pattern is interpreted as both a single precision and a double precision instruction as a function of the precision mode bit.
POP-11 ISP DESCRIPTION 2-1

MACRO BEGIN
$ ( )$

MACRO END
$ ( )$

Imp state

mb([#167777:0]<7:0>)

m[0:#167777:0]<7:0>

mbio([#777777:0]<7:0>)

m[167777:0]<7:0>

mb([#377777:0]<15:8>)

m[0:#377777:0]<7:0>

mbio([#377777:0]<15:8>)

m[0:#377777:0]<7:0>

mar\memory.addrr.reg<17:0>

mar<word.mar<17:1>

mbr\memory.buf.reg<15:0>

mbr<byte.mbr<7:0>

shsign\sh.instruction.offset.sign>

mbr<5:0>

shval\sh.instruction.offset.value<0:0>

mbr<4:0>

!pc state

r\register<7:0><15:0>

sp<15:0>

pc<15:0>

ps<15:0>

! program status word

can\current.mode<1:0>

p\priority<2:0>

trace<>

cc\condition.codes<3:0>

r\negative<>

z\zero<>

o\overflow<>

c\carry<>

a\activity<0:1>

macro run

macro wait

macro halt

! floating point processor state

fppar\fpp.status.register<15:0>

fel<>

fic<>

fluc<>

fivc<>

fppar<10:0>

fppar<9:0>

fppar<8:0>

fppar<7:0>

fppar<6:0>

fppar<5:0>

fppar<4:0>

fppar<3:0>

fppar<2:0>

fppar<1:0>

fppar<0:0>

fppar<15:0>

fppar<14:0>

fppar<13:0>

fppar<12:0>

fppar<11:0>

fppar<10:0>

fppar<9:0>

fppar<8:0>

fppar<7:0>

fppar<6:0>

fppar<5:0>

fppar<4:0>

fppar<3:0>

fppar<2:0>

fppar<1:0>

fppar<0:0>

fppar<15:0>

fppar<14:0>

fppar<13:0>

fppar<12:0>

fppar<11:0>

fppar<10:0>

fppar<9:0>

fppar<8:0>

fppar<7:0>

fppar<6:0>

fppar<5:0>

fppar<4:0>

fppar<3:0>

fppar<2:0>

fppar<1:0>

fppar<0:0>

fppar<15:0>

fppar<14:0>

fppar<13:0>

fppar<12:0>

fppar<11:0>

fppar<10:0>

fppar<9:0>

fppar<8:0>

fppar<7:0>

fppar<6:0>

fppar<5:0>

fppar<4:0>

fppar<3:0>

fppar<2:0>

fppar<1:0>

fppar<0:0>

fppar<15:0>

fppar<14:0>

fppar<13:0>

fppar<12:0>

fppar<11:0>

fppar<10:0>

fppar<9:0>

fppar<8:0>

fppar<7:0>

fppar<6:0>

fppar<5:0>

fppar<4:0>

fppar<3:0>

fppar<2:0>

fppar<1:0>

fppar<0:0>

B-93
POPP-11 ISP DESCRIPTION 2-2

fic<> := fppsr<8>;  //Floating interrupt on integer conversion error enable flag

fcd<> := fppsr<7>;  //Floating precision, one implies double precision, zero single

fl<> := fppsr<6>;  //Integer precision for integer to floating conversions. One implies double precision, zero single.

ft<> := fppsr<5>;  //Truncation or round result. One implies truncation, zero rounding.

fmm<> := fppsr<4>;  //Maintenance mode

fn<> := fppsr<3>;  //Floating negative condition code

fz<> := fppsr<2>;  //Floating zero condition code

f<<> := fppsr<1>;  //Floating overflow condition code

fc<> := fppsr<0>;  //Floating carry condition code
memory management

par\page\address\register\{15:0\}<15:0>
pdr\page\description\register\{15:0\}<15:0>  #: main\{377737:377720\}<15:0>;

macro\pa\page\address\field\{11:8\};
macro\ac\access\control\field\{2:18\}
macro\ed\expansion\direction\{38\}
macro\wbit\written\bit\{68\}
macro\pl\page\length\field\{14:8\}

sr0\status\register\{0<15:0\}  #: main\{3777675\}<15:0>
  an\abort\nonresident\flag\{15\}  #: sr0<15>
  ap\abort\page\length\flag\{14\}  #: sr0<14>
  ar\abort\read\only\flag\{13\}  #: sr0<13>
  am\abort\mode\{8\}  #: sr0<8>
  an\abort\page\number\{2:6\}  #: sr0<2:14>
  e\enable\memory\management\{15\}  #: sr0<15>

sr2\status\register\{2<15:0\}  #: main\{3777677\}<15:0>

B-95
PDP-11 ISP DESCRIPTION 4-1

error flags and temporary registers
----------------------------------
boundary.error;
stack.overflow;
illegal.instruction;
byo$byte.read.flag;
sbyo$save.area.for.byoc;
src<17:0>;
dat<17:0>;
temp<17:0>;
temp<31:0>;
macro dcond := IF dmoed EQL #88
macro scond := IF smoed EQL #88
temp2<32:0>;
PDP-11 ISP DESCRIPTION 5-1

[Instruction format]

\[\text{instruction} < 15:0>\]
\[\text{bop: binary-operation} < 2:0> \quad := 14:12;\]
\[\text{ir: instruction.register} < 15:0> := 15:8;\]

[Source addressing information]
\[\text{source.field} < 5:0> := 11:6;\]
\[\text{sm: source.mode} < 1:0> := 5:4;\]
\[\text{sd: source.deferred} := 3;\]
\[\text{sr: source.register} < 2:0> := 2:0;\]

macro ar67 := (sr < 2:1> eq #3) $

[Destination addressing information]
\[\text{destination.field} < 5:0> := 5:0;\]
\[\text{dm: destination.mode} < 1:0> := 5:4;\]
\[\text{dd: destination.deferred} := 3;\]
\[\text{dr: destination.register} < 2:0> := 2:0;\]

macro dr67 := (dr < 2:1> eq #3) $

[Unary operation]
\[\text{up: unary.operation} < 2:0> := 8:6;\]
\[\text{offset} < 7:0> := 7:0;\]
\[\text{rop: register.operation} < 1:0> := 7:0;\]
\[\text{jsop: jsr.emulator.trap.op} := 15:7;\]
\[\text{etop: emulator.trap.op} := 8;\]
\[\text{conco: condition.code.op} < 10:0> := 15:5;\]
\[\text{cpuop: cpu.control.op} < 2:0> := 2:0;\]
\[\text{contop: cpu.control.class.op} < 2:0> := 5:3;\]
\[\text{broe: branch.op} < 1:0> := 10:8;\]
\[\text{intop: extended.integer.op} < 2:0> := 11:9;\]
\[\text{typeop: class.op} < 4:0> := 10:6;\]
\[\text{resop: reserve.op} := 11;\]
\[\text{ccep: condition.code.second.op} := 4;\]

[Floating point instruction decoding]
\[\text{fbop: floating.binary.operation} < 3:0> := 11:8;\]
\[\text{fup: floating.unary.operation} < 1:0> := 7:6;\]
\[\text{fmsop: floating.mode.setting.op} < 1:0> := 11:8;\]
\[\text{fdsop: floating.double.single.mode.setting.op} := 14;\]

[End of register declarations]
POP-11 ISP DESCRIPTION 6-1

---

# functional declarations

---

abort :=
BEGIN
am ← cm; apn ← mar<15:13> NEXT
pc ← #258 END;

read :=
BEGIN
(IF (eqn a)
  (DECODE cm ← temp ← mar<15:13> abort; abort; temp ← (mar<15:13><0><3:0>) NEXT
   mar ← ((par<temp<psf> ← mar<12:6> ← mar<11:0> ← mar<5:0> NEXT
   (IF not pdr<temp<act> ← abort; anr ← 1) NEXT
   (IF (mar<12:6> gtr pdr<temp<psf>) and NOT pdr<temp<cod> ← abort; aple ← 1) NEXT
   (IF (mar<12:6> iss pdr<temp<psf>) and pdr<temp<cod> ← abort; aple ← 1)

IF mar<15:13> EQL #7 ← mar<17:16> ← #3) NEXT  ! map into lo page
(DECODEx mar<17:13> eqn #37 ←
\no  (DECODE byop ← mbr ← mar<0:mar> ← mbr ← mb<mar>
\yes  (DECODE byop ← mbr ← mar<0:mar> ← mbr ← mb<mar>)
)
END;

write :=
BEGIN
(IF (eqn a)
  (DECODE cm ← temp ← mar<15:13> abort; abort; temp ← (mar<15:13><0><3:0>) NEXT
   mar ← ((par<temp<psf> ← mar<12:6> ← mar<11:0> ← mar<5:0> NEXT
   (IF pdr<temp<act> ← eqn 0 ← abort; anr ← 1) NEXT
   (IF pdr<temp<act> ← eqn 1 ← abort; anr ← 1) NEXT
   (IF (mar<12:6> gtr pdr<temp<psf>) and NOT pdr<temp<cod> ← abort; aple ← 1) NEXT
   (IF (mar<12:6> iss pdr<temp<psf>) and pdr<temp<cod> ← abort; aple ← 1) NEXT
   pdr<temp<psf> ← 1
)

IF mar<15:13> EQL #7 ← mar<17:16> ← #3) NEXT
(DECODEx mar<17:13> eqn #37 ←
\no  (DECODE byop ← mbr ← mar<0:mar> ← mbr ← mb<mar>
\yes  (DECODE byop ← mbr ← mar<0:mar> ← mbr ← mb<mar>)
)
END;

bus.reset := (pc ← pc);
nop := (temp ← temp);

---

B-98
Operand determination

| Source loads the value of the source operand into register src. |
| Dest loads the address of the destination operand into register dst and fetches the operand to the mbr. |

**Source :=**

```plaintext
BEGIN
  (DECODE sm =>
    ASREG\8:src = r[src] NEXT
  )
  (DECODE sd =>
    \0 SREGD=mnop
    \1 SREGD=mnop
  )
END;
```

**ASINC\1:=BEGIN**

```plaintext
  (DECODE ed =>
    \0 (DECODE byop =>
      \0 SINC=mnop
      \1 SINC=mnop
    )
    \1 SINC=mnop
  )
  NEXT
  mar = r[src] NEXT
  (DECODE src87 or sd => r[src] = (r[src] + (2-byop))-15; b[r[src] + (r[src] + 2) < 15]) NEXT
  read NEXT
  src = mbr
END;
```

**ASDEC\2:=BEGIN**

```plaintext
  (DECODE ed =>
    \0 (DECODE byop =>
      \0 SDEC=mnop
      \1 SDEC=mnop
    )
    \1 SDEC=mnop
  )
  NEXT
  (DECODE src87 or sd => r[src] = (r[src] - 2-(byop))<15; b[r[src] + (r[src] - 2) < 15]) NEXT
  read NEXT
  src = mbr
END;
```

**ASINO\3:=BEGIN**

```plaintext
  (DECODE ed =>
    \0 SINO=mnop
    \1 SINO=mnop
  )
  NEXT
  mar = pc NEXT
  pc = (pc + 2)<15> NEXT
  read NEXT
  mar = fbr + r[src]<15> NEXT
  read NEXT
  src = mbr
END
```

**Dest :=**

```plaintext
BEGIN
  (IF sd => mar = src NEXT read NEXT src = mbr )
END;
```

**B-99**
---

**POP-11 ISP DESCRIPTION 7-2**

```plaintext

(DECODE dd =>
  |general registers have addresses 777700:777717
ADREG\0=(dst = (37400 @ dr)) NEXT
  (DECODE dd =>
    \0 DREGs=nop;
    \1 DREGs=nop)
  );

ADINCR\1=BEGIN
  (DECODE dd =>
    \0 (DECODE byop =>
      \0 DINC\1s=nop;
      \1 DINC\1s=nop
    );
    \1 DINC\2s=nop
  ) NEXT
dst = r[dr] NEXT
  (DECODE dr=07 or dd => r[dr] = (r[dr]+(2-byop))<15:8>; r[dr] = (r[dr]+2)<15:8>)
END;

ADDEC\2=BEGIN
  (DECODE dd =>
    \0 (DECODE byop =>
      \0 ODECR\2s=nop;
      \1 ODECR\2s=nop
    );
    \1 ODECR\3s=nop
  ) NEXT
  (DECODE dr=07 or dd => r[dr] = (r[dr]+(2-byop))<15:8>; r[dr] = (r[dr]+2)<15:8>) NEXT
dst = r[dr] END;

ADIND\3=BEGIN
  (DECODE dd =>
    \0 DINDs=nop;
    \1 DIND\3s=nop
  ) NEXT
  mar = pc NEXT
  pc = (pc + 2)<15:0> NEXT
  read NEXT
dst = (mbr + r[dr])<15:0>
END
  ) NEXT
mar = dst NEXT

(IF dd => sbyop = byop NEXT byop = 0 NEXT read NEXT byop = sbyop; dst = mbr; mar = mbr)
END;
```

---
POP-11 ISP DESCRIPTION 8-1

|condition code setting and branch operations|

setnc: \texttt{set.n.condition.code}:
\begin{verbatim}
  (DECODE byop => n + temp<15>; n + temp<7>);
\end{verbatim}

setvc: \texttt{set.v.condition.code}:
\begin{verbatim}
  (DECODE byop => v + (temp<15:0> eqi #00000000); v + (temp<7:0> eqi #200));
\end{verbatim}

setz: \texttt{set.z.condition.code}:
\begin{verbatim}
  (DECODE byop => z + (temp<15:0> eqi #0); z + (temp<7:0> eqi #0));
\end{verbatim}

signextend:
\begin{verbatim}
  (DECODE offset<7> => temp + offset; temp + #377 @ offset);
\end{verbatim}

branch:
\begin{verbatim}
BEGIN
  signextend NEXT
  pc + (pc + (temp 7:0)<15:0>
END;
\end{verbatim}
subroutine, emulator, trap, and trap instructions:
BEGIN
DECODE jetop =>

! jump to subroutine, jsr op code #004
JSR\0: BEGIN
dest NEXT
temp = mar NEXT
sp = (sp - 2)<15:0> NEXT
mar = sp; mbr = r(ar) NEXT
write NEXT
r(ar) = pc NEXT
pc = temp<15:0>
END;

\1 BEGIN
DECODE i<8 =>
! emulator trap op codes, op code #104000:#104377
EMUL\0: BEGIN
byop = 0; sp = (sp - 2)<15:0>; mbr = ps NEXT
write NEXT
sp = (sp - 2)<15:0>; mbr = pc NEXT
write NEXT
mar = #38 NEXT
read NEXT
pc = mbr NEXT
mar = #32 NEXT
read NEXT
ps = mbr
END;

! trap op codes, op code #104400:#104777
TRAP\1: BEGIN
byop = 0; sp = (sp - 2)<15:0>; mbr = ps NEXT
write NEXT
sp = (sp - 2)<15:0>; mbr = pc NEXT
write NEXT
mar = #34 NEXT
read NEXT
pc = mbr NEXT
mar = #36 NEXT
read NEXT
ps = mbr
END
END;

single operand, instructions:
BEGIN
DECODE uop =>

! clear and clear byte, clr op code #0850, clrb op code #1050
ICLR\0: BEGIN
(DECODE byop =>
( CLR=nop NEXT (dcond => (dclrb=nop));
( CLRB=nop NEXT (dcond => (dclrb=nop));

ccir:
ICC = '8100) NEXT
dest NEXT
mbr = 0 NEXT
write
END;

END;
CO: complement byte, com op code #0051, comb op code #1051  
ICOM(I:= BEGIN  
(DECODE byop =>  
( COM:=nop NEXT (dcond => (dcom:=nop))));  
( COM:=nop NEXT (dcond => (dcomb:=nop)))  
) NEXT  
deal NEXT  
temp = not mbr NEXT  
Ccom:=  
(v>0 NEXT  
c=1) NEXT  
setnc NEXT  
setzcc NEXT  
mbr + temp<15:8> NEXT  
write  
END;

IINC2:= BEGIN  
(DECODE byop =>  
( INC:=nop NEXT (dcond => (dinc:=nop))));  
( INCB:=nop NEXT (dcond => (dincb:=nop)))  
) NEXT  
deal NEXT  
temp = mbr + 1 NEXT  
setvcc NEXT  
setnc NEXT  
setzcc NEXT  
mbr + temp<15:8> NEXT  
write  
END;

IDEC3:= BEGIN  
(DECODE byop =>  
( DEC:=nop NEXT (dcond => (ddac:=nop))));  
( DECB:=nop NEXT (dcond => (ddacb:=nop)))  
) NEXT  
deal NEXT  
temp = mbr - 1 NEXT  
setvcc NEXT  
setnc NEXT  
setzcc NEXT  
mbr + temp<15:8> NEXT  
write  
END;

INEG4:= BEGIN  
(DECODE byop =>  
( NEG:=nop NEXT (dcond => (dneg:=nop))));  
( NEGB:=nop NEXT (dcond => (dnegb:=nop)))  
) NEXT  
deal NEXT  
temp = (not mbr) + 1 NEXT  
setvcc NEXT  
setnc NEXT  
setzcc NEXT  
Cneg:=  
(c = (temp<15:8> neq 0)) NEXT

B-103
mbr = temp<15:0> NEXT
write END;

! add carry and add carry byte, adc op code #8055, adcb op code #1055

ADC\5:= BEGIN
(DECODEx byop =>
( ADC:=nop NEXT (dcond => (dadc:=nop)));
( ADCB:=nop NEXT (dcond => (dadcbs:=nop))) ) NEXT
dest NEXT
read NEXT
temp = mbr + c NEXT
cadc:= (DECODEx byop =>
\0 BEGIN
v = (temp<15:0> eql #100000) and c NEXT
c = ( (temp<15:0> eql 0) and c )
END;
\1 BEGIN
v = (temp<7:0> eql #200) and c NEXT
c = ( (temp<7:0> eql 0) and c )
END
) NEXT
setncc NEXT
setzcc NEXT
mbr = temp<15:0> NEXT
write END;

! subtract and subtract carry byte, sbc op code #8056, sbcb op code #1056

SBC\6:= BEGIN
(DECODEx byop =>
( SBC:=nop NEXT (dcond => (dsbc:=nop)));
( SBCB:=nop NEXT (dcond => (dsbcbs:=nop))) ) NEXT
dest NEXT
read NEXT
temp = mbr - c NEXT
csbc:= (DECODEx byop =>
c = ( (temp<15:0> eql #177777) and c )
c = ( (temp<7:0> eql #277) and c )
) NEXT
setvcc NEXT
setncc NEXT
setzcc NEXT
mbr = temp<15:0> NEXT
write END;

! test and test byte, tst op code #0057, tstb op code #1057

TST\7:=BEGIN
(DECODEx byop =>
( TEST:=nop NEXT (dcond => (dttst:=nop)));
( TESTB:=nop NEXT (dcond => (dttstb:=nop))) ) NEXT
dest NEXT
read NEXT
temp = mbr NEXT
tstt:= (v=0 NEXT
c=0) NEXT
setncc NEXT
setzcc
write END;

B-104
POP-11 ISP DESCRIPTION 9-4

END;

shiftop\shift instrucrions:=
BEGIN
DECODE uop =>
! rotate right and rotate right byte, ror op code #0000, rorb op code #1000
IROR\10\0=
BEGIN
(DECODE byop =>
( ROR=nop NEXT (dcond => (drot=nop)));
( RORB=nop NEXT (dcond => (drotb=nop)));
) NEXT
dest NEXT
read NEXT
temp = mbr NEXT
(DECODE byop =>
\0 (temp<16:0> = (c & temp<15:0>) trr 1 NEXT crol:=( c = temp<16>); mbr = temp<15:0>);
\1 (temp<8:0> = (c & temp<7:0>) trr 1 NEXT crolb:=( c = temp<8>); bmb = temp<7:0>)
) NEXT
setncc NEXT
setzcc NEXT
cvror:=
(v = n xor c) NEXT
write
END;

! rotate left and rotate left byte, rol op code #0001, rolb op code #1001
IROL\10\1=
BEGIN
(DECODE byop =>
( ROL=nop NEXT (dcond => (drol=nop)));
( ROLB=nop NEXT (dcond => (drolb=nop)));
) NEXT
dest NEXT
read NEXT
temp = mbr NEXT
(DECODE byop =>
(trol= (temp<16:0> = (c & temp<15:0>) trr 1 NEXT crol:=( c = temp<16>); mbr = temp<15:0>);
(trolb= (temp<8:0> = (c & temp<7:0>) trr 1 NEXT crolb:=( c = temp<8>); bmb = temp<7:0>)
) NEXT
setncc NEXT
setzcc NEXT
cvrol:=
(v = n xor c) NEXT
write
END;

! arithmetic shift right and arithmetic shift right byte, asr op code #0002, asrb op code #1002
INSR\20\2=
BEGIN
(DECODE byop =>
( ASR=nop NEXT (dcond => (dasr=nop)));
( ASRB=nop NEXT (dcond => (dasrb=nop)));
) NEXT
dest NEXT
read NEXT
temp = mbr NEXT
(DECODE byop =>
(tasr= (temp<15:0> = (temp<15:0>) tarr temp<15> NEXT mbr = temp<15:0>);
(tasrb= (temp<7:0> = (temp<7:0>) tarr temp<7> NEXT mbr = temp<7:0>)
) NEXT
setncc NEXT
setzcc NEXT
cvsrc:=
(v = n xor c) NEXT
write
END;

B-105
! arithmetic shift left and arithmetic shift left byte, asl op code #0003, aslb op code #1003

IRSL\3: = BEGIN
  (DECODE byop =>
    ( RSL: =nop NEXT (dcond => (dest1 =nep)));
    ( PSLB: =nop NEXT (dcond => (dest1 =nep)))
  ) NEXT
  dest NEXT
  read NEXT
  temp + mbr NEXT
  (DECODE byop =>
    (temp<16:0> = (c & temp<15:0>); tel0 NEXT casls:(c = temp<10>); mbr = temp<15:0>);
    (temp<8:0> = (c & temp<7:0>); tel0 NEXT casls:(c = temp<8>); mbr = temp<7:0>))
  ) NEXT
  setnco NEXT
  setnc NEXT
  (v = n xor c) NEXT
  write
END;

! mark and unused op codes, mark op code #0004
MARK\4: = BEGIN
  IF not jtop =>
    sp = (sp + (d tel 0))<15:0> NEXT pc = r(s) NEXT
    mar = sp NEXT
    read NEXT
    r(s) = mbr, sp = (sp + 2)<15:0>
  END;

! move from previous instruction and data space, mfp op code #0005, mfpd op code #1005
MFP\5: = (dcond => (src =nep)) NEXT
  (DECODE jtop => nop<nop>);

! move to previous instruction and data space, mtp op code #0006, mtpd op code #1006
MTP\6: = (dcond => (src =nep)) NEXT
  (DECODE jtop => nop<nep>);

! sign extend and unused op code, sxt op code #0007
SXT\7: = BEGIN
  IF not jtop =>
    (dcond => (dest =nep)) NEXT
    dest NEXT
    read NEXT
    (DECODE n => mbr + 0; mbr = #177777) NEXT
  END
  csxt: =
    (z = not n; v = 0) NEXT
    write
END;

B-106
POP-11 ISP DESCRIPTION 10-1

! condition code operators. selectively clears
! or sets the specified condition code.
! the assembler recognizes the mnemonics
! cic, clv, clz, clm, ccc (for clear all
! condition codes), sec, sev, sez, sen, and scc.
! compound setting or clearing is accomplished
! by oring.

ccc :=
BEGIN
IF (cconop eqi #0005) =>
  (DECODE cconop =>
    \0 cc := cc and not (c<3:0>)
    \1 cc := cc or (c<3:0>)
  )
END;

cpucon\cpu.control.instructions :=
BEGIN
IF contop eqi 8 =>
BEGIN
DECODE cpuop =>
  \halt, halt op code #000000
  HLT\8\a : a = 2;

  \wait for interrupt, wait op code #000001
  EWAIT\1\i = a = 1;

  \return from interrupt, rti op code #000002
  RTI\2\i = BEGIN
    mar = sp NEXT
    read NEXT
    pc = mbr; sp = (sp + 2)\<15:0> NEXT
    mar = sp NEXT
    read NEXT
    ps = mbr; sp = (sp + 2)\<15:0>
    END;

  \breakpoint trap, bpt op code #000003
  BPT\3\i = BEGIN
    sp = (sp - 2)\<15:0>; mbr = ps NEXT
    write NEXT
    sp = (sp - 2)\<15:0>; mbr = pc NEXT
    write NEXT
    mar = \#14 NEXT
    read NEXT
    pc = mbr NEXT
    mar = \#16 NEXT
    read NEXT
    ps = mbr
    END;

  \input/output trap, iot op code #000004
  IOT\4\i = BEGIN
    sp = (sp - 2)\<15:0>; mbr = ps NEXT
    write NEXT
    sp = (sp - 2)\<15:0>; mbr = pc NEXT
    write NEXT
    mar = \#20 NEXT
    read NEXT
    pc = mbr NEXT
    mar = \#22 NEXT
B-107
read NEXT
ps = mbr
END;

! reset external bus, reset op code #000005
RSET

! return from trap, rti op code #000006
RTI = BEGIN
mar = sp NEXT
read NEXT
pc = mbr; sp = (sp + 2)<15:0> NEXT
mar = sp NEXT
read NEXT
ps = mbr; sp = (sp + 2)<15:0>
END;

! unused op code
\7 nop
END

procon\program.control.instructions :=
BEGIN
DECODE contop =>
! return from subroutine, rts op code #000020
RTS\8 = BEGIN
pc = r{dr} NEXT
mar = sp NEXT
read NEXT
sp = (sp + 2)<15:0> NEXT
r{dr} = mbr
END;

! unused op code
\1 nop;

! unused op code
\2 nop;

! set priority level, spl op code #000023
! 11/70 instruction
SPL\3 = nop;

! NEXT four op codes are condition code setting
\4 cco;
\5 cco;
\6 cco;
\7 cco
END;
regop\register operations:: 
BEGIN
DECODE rop =>
\cpu control instructions
\0 cpucon;

\jump, jmp op code #0801
JMP\11 BEGIN
dest NEXT
pc = mar<15:0>
END;

\ program control instructions
\2 procon;

\ swap bytes, swab op code #0003
SHAB3$ BEGIN
\(cond => (dswab=\nop)) NEXT
dest NEXT
read NEXT
temp = bmbr & mbr<15:0> NEXT
\swap:=
\(n = temp<7:0> z = (temp<7:0> eql 0));
v = 0; c = 0) NEXT
write
END
END;
branch_op_codes:
BEGIN
  DEC0DE JP op eq br =
  ! register instructions
  \0 regop;
  branch, br op code #0004
  BR\1: = branch;
  branch IF not equal, bne op code #0010
  BNE\2: = (IF not z => branch);
  branch IF equal, beq op code #0014
  BEQ\3: = (IF z => branch);
  branch IF greater than or equal, bge op code #0020
  BGE\4: = (IF not (n xor v) => branch);
  branch IF less than, blt op code #0024
  BLT\5: = (IF (n xor v) => branch);
  branch IF greater than, bgt op code #0030
  BGT\6: = (IF not (z or (n xor v)) => branch);
  branch IF less than or equal, ble op code #0034
  BLE\7: = (IF (z or (n xor v)) => branch);
  branch IF plus, bpl op code #1000
  BPL\10: = (IF not n => branch);
  branch IF minus, bmi op code #1004
  BM1\11: = (IF n => branch);
  branch IF higher, bhi op code #1010
  BHI\12: = (IF not c and (not z) => branch);
  branch IF lower or equal, ble op code #1014
  BLO\13: = (IF (c or z) => branch);
  branch IF overflow clear, bvc op code #1020
  BVC\14: = (IF not v => branch);
  branch IF overflow set, bvs op code #1024
  BVS\15: = (IF v => branch);
  branch IF carry clear, bcc op code #1030
  BCC\16: = (IF not c => branch);
  branch IF carry set, bcs op code #1034
  BCS\17: = (IF c => branch)
END;
POP-11 ISP DESCRIPTION 13-1

integer.extended.op.codes =
BEGIN
DECODE intop =>
  ! integer multiply, mul op code #070
  ! 11/40 extended instruction
MUL\01 = BEGIN
  (dcond => (muln=not)) NEXT
  dest NEXT
  read NEXT
  (DECODE mbr<15> =>
    \0 (DECODE r[ar]<15> =>
      \0 temp2=(mbr[ar])<31:0> NEXT
      \1 (temp2=(mbr[MINUS r[ar]])<31:0> NEXT
        temp2=(MINUS temp2)<32:0>)
    );
    \1 (DECODE r[ar]<15> =>
      \0 (temp2=(MINUS mbr)[ar]<31:0> NEXT
        temp2=(MINUS temp2)<32:0>)
      \1 (temp2=(MINUS mbr)[MINUS r[ar]]<31:0>)
    )
  )
) NEXT

! integer divide, div op code #071
! 11/40 extended instruction
DIV\1 = BEGIN
  (dcond=(adi=not)) NEXT
  dest NEXT
  read NEXT
  (IF (mbr NEQ 0) =>
    (DECODE mbr<15> =>
      \0 (DECODE r[ar]<15> =>
        \0 (temp2=(r[ar][ar][ar OR #1])<31:0>/mbr NEXT
          r[ar OR #1]=(r[ar][ar][ar OR #1]-temp2<15:0>/mbr)<15:0> NEXT
          v=(r[ar] GTR mbr)
        );
        \1 (temp2=(MINUS r[ar][ar][ar OR #1])<31:0>/mbr NEXT
          r[ar OR #1]=(MINUS (r[ar][ar][ar OR #1]-temp2<15:0>/mbr))<15:0> NEXT
          temp2=(MINUS temp2)<32:0> NEXT
          v=(MINUS r[ar] GTR mbr)
        )
      );
      \1 (DECODE r[ar]<15> =>
        \0 (temp2=(r[ar][ar][ar OR #1])<31:0>/(MINUS mbr) NEXT
          r[ar OR #1]=(MINUS (r[ar][ar][ar OR #1]-temp2<15:0>/MINUS mbr))<15:0> NEXT
          temp2=(MINUS temp2)<37:0> NEXT
          v=(r[ar] GTR (MINUS mbr))
        );
        \1 (temp2=(MINUS r[ar][ar][ar OR #1])<31:0>/(MINUS mbr) NEXT
          r[ar OR #1]=(MINUS r[ar][ar][ar OR #1]-temp2<15:0>/MINUS mbr)<15:0> NEXT
          v=(MINUS r[ar] GTR MINUS mbr)
        )
      )
    )
  )
) NEXT

! integer divide, div op code #071
! 11/40 extended instruction
DIV\1 = BEGIN
  (dcond=(adi=not)) NEXT
  dest NEXT
  read NEXT
  (IF (mbr NEQ 0) =>
    (DECODE mbr<15> =>
      \0 (DECODE r[ar]<15> =>
        \0 (temp2=(r[ar][ar][ar OR #1])<31:0>/mbr NEXT
          r[ar OR #1]=(r[ar][ar][ar OR #1]-temp2<15:0>/mbr)<15:0> NEXT
          v=(r[ar] GTR mbr)
        );
        \1 (temp2=(MINUS r[ar][ar][ar OR #1])<31:0>/mbr NEXT
          r[ar OR #1]=(MINUS (r[ar][ar][ar OR #1]-temp2<15:0>/mbr))<15:0> NEXT
          temp2=(MINUS temp2)<32:0> NEXT
          v=(MINUS r[ar] GTR mbr)
        )
      );
      \1 (DECODE r[ar]<15> =>
        \0 (temp2=(r[ar][ar][ar OR #1])<31:0>/(MINUS mbr) NEXT
          r[ar OR #1]=(MINUS (r[ar][ar][ar OR #1]-temp2<15:0>/MINUS mbr))<15:0> NEXT
          temp2=(MINUS temp2)<37:0> NEXT
          v=(r[ar] GTR (MINUS mbr))
        );
        \1 (temp2=(MINUS r[ar][ar][ar OR #1])<31:0>/(MINUS mbr) NEXT
          r[ar OR #1]=(MINUS r[ar][ar][ar OR #1]-temp2<15:0>/MINUS mbr)<15:0> NEXT
          v=(MINUS r[ar] GTR MINUS mbr)
        )
      )
    )
  )
) NEXT

B-111
.cddiv:= (v=(mbr EQL 0) OR v) NEXT
ccddiv:= (c=(mbr EQL 0)) NEXT
r(sr)=temp2<15:0>
END;

! shift arithmetically, ash op code #072
ASH\2:= BEGIN
(dcond => (dashc=nop)) NEXT
dest NEXT
read NEXT
temp += r(sr) NEXT
BEGIN
  DEC\DE shsign =>
  /0 (temp<15:0>=cgt(temp<15:0>) ts10 shval NEXT
  (c=temp<15:0>));
  \1 (DEC\DE temp<15> =>
    /0 temp<15:0>=temp<15:0> tar0 (not shval)+1);
    \1 temp<15:0>=temp<15:0> tar1 (not shval)+1) NEXT
  (c=temp<8>));
.
cash:=
END NEXT
setnc NEXT
setzc NEXT
cashb:=
(v=temp<15> xor mbr<15>) NEXT
r(sr)=temp<15:0>
END;

! arithmetical shift combined, as\hc op code #073
LI/70 extended instruction
ASHC\3:= BEGIN
(dcond => (dashc=nop)) NEXT
dest NEXT
read NEXT
temp += r(sr) NEXT
BEGIN
  DEC\DE shsign =>
  /0 (temp<32:0>=cgt(temp<15:0>sr{sr OR #1})) ts10 shval NEXT
  (c=temp<16>));
  \1 (DEC\DE temp<15> =>
    /0 temp<31:0>=temp<15:0>sr{sr OR #1} tar0 (not shval)+1);
    \1 temp<31:0>=temp<15:0>sr{sr OR #1} tar1 (not shval)+1) NEXT
  (c=temp<32:0>));

  cas\chc:=
END;

\1 BEGIN
  DEC\DE shsign =>
  /0 (temp<16:0>=cgt(temp<15:0>) ts10 shval NEXT
  (c=temp<15:0>));
  \1 (DEC\DE temp<15> =>
    /0 temp<15:0>=temp<15:0> tar0 (not shval)+1);
    \1 temp<15:0>=temp<15:0> tar1 (not shval)+1) NEXT
  (c=temp<16:0>));

cnashc:=
END) NEXT
(c\DE shrc sr<0> => n-temp<31:0>; n=temp<15:0>) NEXT
c\ashc:=
(c\DE shrc sr<0> => z=(temp<31:0> eq sl)); z=(temp<15:0> eq o) ) NEXT
c\oshc:=
(c\DE shrc sr<0> => v=(temp<31:0> xor mbr<15>)); v=(temp<15:0> xor mbr<15>) NEXT
(DEC\DE sr<0> => (sr{sr}+temp<31:16> NEXT r{sr OR #1}=temp<15:0>); (r{sr}+temp<15:0>))
END;

! exclusive or, xor op code #074
EX\2:=BEGIN
(dcond => (d\xorc=nop)) NEXT
dest NEXT
read NEXT

PDP-11 ISP DESCRIPTION 13-3

temp = r[ar] xor abr NEXT
abr = temp NEXT
setnc NEXT
setznc NEXT

cc xor :=
   (v = 0) NEXT
write
END;

| remaining instructions 11/40 floating point or unused op codes |
|---|---|
| \5| nop; |
| \6| nop; |

SOB7 := BEGIN
   r[ar] := (r[ar] - 1) \times 15:0 NEXT
   (IF r[ar] = 0 => pc := (pc - 1) \times 15:0 \times 15:0)
END;

farcon\floating.point.processor.mode,control :=
<DECODE fsoop =>
\0 BEGIN
DECODE fsoop =>
   ! copy floating condition codes, cfcc
   ! op code #170000
   cfcc'0 := nop;
   ! set single precision floating mode, ssof
   ! op code #170001
   ssof'1 := nop;
   ! set single precision integer mode, set1
   ! op code #170002
   set1'2 := nop;
   ! unused op code #170003
\3 nop
END;

\1 BEGIN
DECODE fsoop =>
   ! unused op code #170010
\0 nop;
   ! set double precision floating mode,
   ! setd op code #170011
   setd'1 := nop;
   ! set double precision integer mode, setd
   ! op code #170012
   setd'2 := nop;
   ! unused op code #170013
\3 nop
END

fsingle\floating.point.single.operand.instructions :=
R-173
POP-11 ISP DESCRIPTION 13-4

(DECODER id =>
  \0 BEGIN
  DECODE fuop =>
    \ 1 clear floating, clrf op code #1704
  clrf\0 := (dcond => dclrf=nop) NEXT dest;

    \ 2 test floating, tstf op code #1705
  tstf\1 := (dcond => dstf=nop) NEXT dest;

    \ 3 form absolute value, absf op code #1706
  absf\2 := (dcond => dabst=nop) NEXT dest;

    \ negate floating, negf op code #1707
  negf\3 := (dcond => dnegf=nop) NEXT dest)
END;

(DECODER fd =>
  \0 BEGIN
  DECODE fuop =>
    \ 1 clear floating, clrd op code #1704
  clrd\0 := (dcond => dclrd=nop) NEXT dest;

    \ 2 test floating, tstd op code #1705
  tstd\1 := (dcond => dstd=nop) NEXT dest;

    \ 3 form absolute value, absd op code #1706
  absd\2 := (dcond => dabst=nop) NEXT dest;

    \ negate floating, negd op code #1707
  negd\3 := (dcond => dnegd=nop) NEXT dest)
END
)

fppcon\floating.point.processor.control :=
(DECODER fuop =>
  \ 0 floating status register setting instructions

    \ load fpp processor status word, ldfps op code #1701
  ldfps\1 := (dcond => dldfps=nop) NEXT dest;

    \ store fpp processor status word, stfps op code #1702
  stfps\2 := (dcond => dstfps=nop) NEXT dest;

    \ store fpp status including exception address pointer,
  \ stat op code #1703
  stat\3 := (dcond => dstat=nop) NEXT dest)
)

fpe\x\floating.point.processor.isp:=
(DECODER fd =>
  \0 BEGIN
  DECODE fuop =>
    \ floating point processor mode control

    \ floating point unary instructions

B-114
\1 (single;

! floating multiply, multi op code \#1710x
multi\2 := ( (dcond -> dmulti: = nop) NEXT dest);

! multiply and integerize floating, mod\f op code \#1714x
mod\f\3 := ( (dcond -> dmod\fis: = nop) NEXT dest);

! floating add, addf op code \#1720x
addf\4 := ( (dcond -> daddfis: = nop) NEXT dest);

! load floating register, idf op code \#1724x
idf\5 := ( (dcond -> didfis: = nop) NEXT dest);

! floating subtract, subf op code \#1730x
subf\6 := ( (dcond -> dsubfis: = nop) NEXT dest);

! floating compare, cmpf op code \#1734x
cmpf\7 := ( (dcond -> dcmpfis: = nop) NEXT dest);

! store floating register, stf op code \#1740x
stf\10 := ( (dcond -> dstfis: = nop) NEXT dest);

! floating divide, divf op code \#1744x
divf\11 := ( (dcond -> ddivfis: = nop) NEXT dest);

! store floating exponent, stexp op code \#1750x
stexp\12 := ( (dcond -> dstexpis: = nop) NEXT dest);

! store and convert from floating to integer, stc op code \#1754x
 \13 (DECODE fl =>
    stcfl\8 := ( (dcond -> dstcfl: = nop) NEXT dest);
    stcfl\1 := ( (dcond -> dstcfl: = nop) NEXT dest));

! convert from floating single to floating double precision,
! stcdf op code \#1760x
stcdf\14 := ( (dcond -> dstcdfis: = nop) NEXT dest);

! load floating exponent, ldexp op code \#1764x
ldexp\15 := ( (dcond -> didexpis: = nop) NEXT dest);

! load and convert from integer to floating, ldc op code \#1770x
 \16 (DECODE fl =>
    ldcfl\8 := ( (dcond -> didcfl: = nop) NEXT dest);
    ldcfl\1 := ( (dcond -> didcfl: = nop) NEXT dest));

! load and convert from floating double to
! floating single, ldcdf op code \#1774x
ldcdf\17 := ( (dcond -> didcdfis: = nop) NEXT dest)

END;

BEGIN
DECODE f8op =>
! floating point processor mode control
\0 fppcon;

! floating point unary instructions
\1 fopng;

! floating multiply, multi op code \#1710x
mulp2 := (dcond => dmulp = nop) NEXT dest);

! multiply and integerize floating, mulp op code #1714x
mulp3 := (dcond => dmulp = nop) NEXT dest);

! floating add, addd op code #1720x
addd4 := (dcond => daddds = nop) NEXT dest);

! load floating register, ldd op code #1724x
ldd5 := (dcond => dlldds = nop) NEXT dest);

! floating subtract, subd op code #1730x
subd6 := (dcond => dsubds = nop) NEXT dest);

! floating compare, cmpd op code #1734x
cmpd7 := (dcond => dccmpds = nop) NEXT dest);

! store floating register, std op code #1740x
std8 := (dcond => dstds = nop) NEXT dest);

! floating divide, divd op code #1744x
divd9 := (dcond => ddivds = nop) NEXT dest);

! store floating exponent, stexp op code #1750x
stexp10 := (dcond => dstexpds = nop) NEXT dest);

! store and convert from floating to integer, stc op code #1754x
\13 (DECODE fl =>
  stcdl0 := (dcond => dstcdl = nop) NEXT dest);

  stcdl1 := (dcond => dstcdl = nop) NEXT dest));

! convert from floating double to floating single precision,
! stcdf op code #1768x
stcdf14 := (dcond => dstcdf = nop) NEXT dest);

! load floating exponent, ldexp op code #1764x
ldexp15 := (dcond => dldexp = nop) NEXT dest);

! load and convert from integer to floating, ldci op code #1770x
\16 (DECODE fl =>
  ldcdl0 := (dcond => dllcdl = nop) NEXT dest);

  ldcdl1 := (dcond => dllcdl = nop) NEXT dest));

! load and convert from floating single to
! floating double, ldcf op code #1774x
ldcdf17 := (dcond => dllcdf = nop) NEXT dest)

END
classop\secondary.decode.into.classes:
BEGIN
DECODE typsop =>
| 1 subroutine/emulator trap | 0  | subent; |
| 1 single operand class      | 1  | singlop; |
| 1 shift operators           | 2  | shiftop; |
| 1 unused op codes           | 3  | nop     |
END;

extop\extended.op.codes:
BEGIN
DECODE jetop =>
| 1 integer extended instructions | 0  | intext; |
| 1 floating point instructions  | 1  | fpext   |
END;

reserop\reserve.op.code:
BEGIN
DECODE resop =>
| 0  | branop; |
| 1  | classop |
END;
exec\instruction.execution=
BEGIN
DECODE byop =>
| reserved op code
\0 reserved
| move and move byte
| mov op code #81, movb op code #11
INMOV1:=
BEGIN
(DECODE byop =>
  ( MOV:=nop NEXT
    (scond => (smov:=nop)) NEXT
    ( dcond => (dmov:=nop)));
  ( MOVb:=nop NEXT
    (scond => (smovb:=nop)) NEXT
    ( dcond => (dmovb:=nop))
  ) NEXT
source NEXT
temp := src NEXT
(v + 0) NEXT
setcc NEXT
setzcc NEXT
dest NEXT
mbc := temp<15:8> NEXT
write
END;

! compare and compare byte
! cmp op code #82
! cmpb op code #12
ICMP2= BEGIN
(DECODE byop =>
  ( CMP:=nop NEXT
    (scond => (scmp:=nop)) NEXT
    ( dcond => (dcmp:=nop)));
  ( CMPb:=nop NEXT
    (scond => (scmpb:=nop)) NEXT
    ( dcond => (dcmpb:=nop))
  ) NEXT
source NEXT
dest NEXT
read NEXT
(DECODE byop =>
\0 temp := (src<15:8> + (NOT mbc) + 1)<16:8>
\0 temp := (src<7:0> + (NOT mbc<7:0>) + 1)<8:0>
) NEXT
setcc NEXT
setzcc NEXT
ccmp:=
(DECODE byop => c := NOT temp<16>; c := NOT temp<8>) NEXT
(DECODE byop =>
\0 v := (temp<15> eqv mbc<15>) and (src<15> xor mbc<15>);
\0 v := (temp<7> eqv mbc<7>) and (src<7> xor mbc<7>)
) END;

! bit test and bit test byte
! bit op code #83, bithop op code #13
IBIT3= BEGIN
(DECODE byop =>
  ( BIT:=nop NEXT
    (scond => (sbith:=nop)) NEXT
    ( dcond => (dbith:=nop)));
  ( BITb:=nop NEXT
    (scond => (sbithb:=nop)) NEXT
  )
B-118
(dcond => (dbits=nop)))
)

NEXT
source NEXT
dest NEXT
read NEXT
temp = src and mbr NEXT
setnce NEXT
setzcc NEXT
(v = 0)

END;

! bit clear and bit clear byte
! bic op code #04, bich op code #14
BIC16s BEGIN

(DECODE byop =>

(BIC=nop NEXT
(scond => (bibc=nop)) NEXT
(dcond => (bibc=nop))
)

NEXT
source NEXT
dest NEXT
read NEXT
temp = (not src<15:0>) and mbr NEXT
setnce NEXT
setzcc NEXT

(v = 0) NEXT
mbr = temp<15:0> NEXT

write

END;

! bit set and bit set byte
! bis op code #05, bisb op code #15
BIS16s BEGIN

(DECODE byop =>

(BIS=nop NEXT
(scond => (bibisb=nop)) NEXT
(dcond => (bibisb=nop))
)

NEXT
source NEXT
dest NEXT
read NEXT
temp = src<15:0> or mbr NEXT
setnce NEXT
setzcc NEXT

(v = 0) NEXT
mbr = temp<15:0> NEXT

write

END;

! add and subtract
\G BEGIN
DECODE byop =>
! add, add op code #06
ADD16s BEGIN

(scond => (addisb=nop)) NEXT
(dcond => (addisb=nop)) NEXT
source NEXT

source NEXT

NEXT
}

NEXT


cbic:

NEXT

write

source NEXT

NEXT


cbic:

NEXT
PDP-11 ISP DESCRIPTION 15-3

! subtract, sub op code #16
SUB\1:= BEGIN
  byop = 0 NEXT
  (cond => (sub=nap)) NEXT
  (dcond => (dsub=nap)) NEXT
  source NEXT
  dest NEXT
  read NEXT
  temp = (mbr + (NOT src)<15:0> +1)<16:0> NEXT
  (v = (src<15> eqv mbr<15>) and (src<15> xor temp<15>) NEXT
  c = NOT temp<16>) NEXT
  setnc NEXT
  setzc NEXT
  mbr = temp<15:0> NEXT
  write NEXT
END;

! extended instruction set
\7 extop
END

ERALCUD
PDP-11 ISP DESCRIPTION 16-1

Main sequence of the ISP description

Instruction interpretation process

Instruction interpretation:

BEGIN
  IF run =>
    mar → pc NEXT
    IF sr0<15:13> && 0 => sr2 = mar<15:8> NEXT
    byop = 0 NEXT
    read NEXT
    ir = mbr; pc = (pc + 2)<15:0> NEXT
    byop = i<15> NEXT
    exec NEXT
    inter
  END

END

End of description
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Sample Simulation Run</td>
<td>2</td>
</tr>
<tr>
<td>2 Simulation Command Files: Benchmark Program</td>
<td>7</td>
</tr>
<tr>
<td>3 Simulation Command Files: Driver Program</td>
<td>9</td>
</tr>
<tr>
<td>4 Simulation Command Files: Unimplemented Instructions</td>
<td>10</td>
</tr>
<tr>
<td>5 Simulation Command Files: Opaqued Procedures</td>
<td>11</td>
</tr>
</tbody>
</table>
1. Sample Simulation Run

The following is a transcript of a typical session using the ISP simulator. The session consists of running one of the benchmarks (Bit Test, Set, and Reset) on the PDP-11. A listing of the benchmark program appears after the session. Comments have been added for clarity.

The input for a simulation session consists of several files prepared off-line. These files include: The benchmark program (derived from the assembly listing), a driver (simulation commands used to initialize the parameters for the benchmark), a command file with a list of unimplemented instructions (these must be trapped), and finally, a command file with a list of those ISP procedures which must be "opaque" (these are the procedures during which the activity counters are disabled).

ru pdp11m
ISP SIMULATOR v3 - NRL AAF STAGE 2
Friday 10 Sep 76 17:13:50 PDP11M.ISP(L410MB25)
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>read fadl.sim Read in the benchmark file
>>RADIX OCTAL
>>DECHO ! The benchmark file disables the listing
on the user terminal.
>>100 LINES READ
>read fa.dr3 ! Read in the driver file
>>! HERE COMES THE DRIVER (CALLS)
>>
>>SETVAL Nh130881-013746 005202 ! MOV @5202,-(SP) ; F
>>SETVAL Nh130821-013746 005204 ! MOV @5204,-(SP) ; N
>>SETVAL Nh130841-013746 004000 ! MOV $4000,-(SP) ; RL
CFA Report

>>SETVAL MN130081-0 012748 005200
>>SETVAL MN130101-0 012748 005200
>>SETVAL MN130121-0 084737 001000
>>SETVAL MN130141-0 000000

The above sequence of POP-11 instructions push the parameters onto the stack, call the benchmark as a routine, and halt.

>>SETVAL MN120081-123457 071234 167000 145670

BIT STRING

>>SETVAL MN125001-0
>>SETVAL MN125011-2
>>SETVAL MN125021-25
>>SETVAL MN125031-0

RETURN CODE
F
N
WORK AREA

>>SETVAL PC-6000
>>SETVAL SP-200

The above sequence initializes the data (parameters), the stack pointer and the program counter (which now points to the code sequence that pushes the parameters and calls the routine.

>>SETVAL H-8

This is an ISP internal variable — indicates whether the machine is running, halted, or waiting.

>>RUNNING

>>SETCTR ALL 0,0

RESET COUNTERS

>>READ OPQ11.SIM(410MB25)

POP11 OPAQUED PROCEDURES

>>D ECHO

>>53 LINES READ

>>READ UUQ11.SIM(410MB25)

UNIMPLEMENTED OPERATION BREAKS

>>D ECHO

>>15 LINES READ

>>TRACE IR,PC,R,MWIO

Trace a few selected registers
IR is the Instruction Register,
PC is the Program Counter (R17),
BREAK JSR, RTS

>26 LINES READ

START

BREAK AFTER JSR

setctr all 8,8

CONT

CFA Report  
October 1, 1976

R(0:7) are the general registers,
NNIO is the I/O page (R is mapped onto NNIO)

! Break on selected instructions

Here we start the simulation

The simulation stops on a breakpoint

The real benchmark starts here, we must
reset all counters (they were modified
during the benchmark calling sequence)

we continue the simulation
BREAK AFTER RTS  // the simulation stops at the end of the
  // benchmark (the return instruction)
  
  #outctr fad1.rw3    // we dump all the counters into a file
  #cont                // we continue the simulation

  #RTS +#2  PC =#1874
  #RTS +#7  R  [#7]=#8938
SIMULATION COMPLETED

we executed the Halt Instruction

RUN TIME (10 usec units) = 831678
RTM OPS EXECUTED = 4535

>exit

we finish the session

EXIT
2. Simulation Command Files: Benchmark Program

RADIX OCTAL
DECHO
ICFAF  MACH11  V003F  5-JUL-76  12:56  PAGE 1
BTSR1  M11

1  00100  .TITLE  CFAF
2  00200  ; Bit test, set, or reset subroutine
3  00300  ; CFAF program F, CMU programmer 3 -
4  00400  ; 8 June 1976
5  00500  .GLOBAL  BTSR
6  00600  RO=Z8
7  00700  SP=Z6
8  00800  PC=Z7
9  00900  ;
10  01000  ; I assume that bits are numbered fr
11  01100  ; of a word.
12  01200  ;
13  01300  ; Offsets of parameters from stack p
14  01400  ;
15  01500  SAVE=4  ; we need to save 2
16  01600  ;
17  01700  F=12+SAVE  ; function code
18  01800  K=16+SAVE  ; relative bit numb
19  01900  R1=8+SAVE  ; address of bit str
20  02000  RC=4+SAVE  ; address of return
21  02100  WORK=2+SAVE  ; address of work ar
22  02200  ;
23  02300  BTSR:  
24  02400  MOV  R8,-(SP)
25  02500  MOV  R1,-(SP)
26  02600  CLR  @R(SP)
27  02700  MOV  N(SP),R8  ; ge
28  02800  BIC  #177778,R8  ; th
29  02900  MOV  #1,R1
30  03000  ASH  R8,R1  ; sh
31  03100  MOV  N(SP),R8  ; by
32  03200  ASH  #3,R8
33  03300  ADD  R1(SP),R8  ; th
34  03400  BITB  R1,@RD
35  03500  BEQ  L1
36  03600  INC  @R(SP)
37  03700  CMP  #2,F(SP)  ; se
38  03800  BEQ  SET
39  03900  BPL  QUIT
40  04000  BICB  R1,@RD  ; FC
41  04100  QUIT:  MOV  (SP)+,R1  ; ex
42  04200  MOV  (SP)+,R8
43  04300  RTS  PC
44  04400  SET:  BSIB  R1,@RD  ; FC
45  04500  BR  QUIT
46  04600  .END
ERRORS DETECTED: 0

# TOTAL # OF PST ACCESSES = 22
# TOTAL # OF 11/45 INSTRUCTIONS = 2
# RUN-TIME:  1 SECONDS
# CORE USED:  4K

! Here begin the simulation commands
! derived from the above listing
! relocation address = word 400 (octal) = byte 1000

| SETVAL  | MH(400) -010046  
| SETVAL  | MH(401) -010146  
| SETVAL  | MH(402) -005076  000010  
| SETVAL  | MH(404) -016600  000014  
| SETVAL  | MH(406) -042700  177770  
| SETVAL  | MH(410) -012701  000001  
| SETVAL  | MH(412) -072100  
| SETVAL  | MH(413) -016600  000014  
| SETVAL  | MH(415) -072027  177775  
| SETVAL  | MH(417) -066600  000012  
| SETVAL  | MH(421) -130118  
| SETVAL  | MH(423) -001482  
| SETVAL  | MH(423) -005276  000010  
| SETVAL  | MH(425) -022766  000002  000016  
| SETVAL  | MH(430) -001405  
| SETVAL  | MH(431) -100001  
| SETVAL  | MH(432) -140110  
| SETVAL  | MH(433) -012601  
| SETVAL  | MH(434) -012600  
| SETVAL  | MH(435) -002287  
| SETVAL  | MH(436) -156110  
| SETVAL  | MH(437) -009773  

ECHO
IV.III.3. Simulation Command Files: Driver Program

1. HERE COMES THE DRIVER (CALLS)

SETVAL MH(30001) - 013740 005202
SETVAL MH(30021) - 013740 005204
SETVAL MH(30041) - 012746 004000
SETVAL MH(30061) - 012746 005200
SETVAL MH(30101) - 012746 005206
SETVAL MH(30121) - 004737 001000
SETVAL MH(30141) - 000000

SETVAL MH(20001) - 123457 071234 167080 145076
SETVAL MH(20001) - 0
SETVAL MH(25001) - 2
SETVAL MH(25021) - 25
SETVAL MH(25031) - 0

SETVAL PC-6000
SETVAL SP-200
SETVAL A-0

MOV @5202,-(SP)
MOV @5204,-(SP)
MOV @4000,-(SP)
MOV @5200,-(SP)
MOV @5206,-(SP)
JSR PC,#1000

HERE COMES THE DRIVER (CALLS)

BIT STRING
RETURN CODE
F
N
WORK AREA

RESET COUNTERS
READ D011.SIM(L410MB25)
READ UU11.SIM(L410MB25)
TRACE IR,PC,R,HW10
BREAK JSR,ATS
IV.III.5. Simulation Command Files: Opaqued Procedures

1 POP11 OPAQUED PROCEDURES
DECHO
OPAQUE CADC
OPAQUE CADD
OPAQUE CASH
OPAQUE CASHB
OPAQUE CASHBC
OPAQUE CASHC1
OPAQUE CASHC2
OPAQUE CASHCB
OPAQUE CASL
OPAQUE CASL8
OPAQUE CASR
OPAQUE CB1C
OPAQUE CB1S
OPAQUE CB1T
OPAQUE CCCO
OPAQUE CCLR
OPAQUE CCMNP
OPAQUE CCM8
OPAQUE CK8R
OPAQUE CM8V
OPAQUE CNASHC
OPAQUE CNEG
OPAQUE CROL
OPAQUE CROLB
OPAQUE CRODR
OPAQUE CRO8B
OPAQUE CSBC
OPAQUE CS8B
OPAQUE CS8AP
OPAQUE CSXT
OPAQUE CTST
OPAQUE CVASH
OPAQUE CVASHC
OPAQUE CVASL
OPAQUE CVASR
OPAQUE CV8L
OPAQUE CV8R
OPAQUE   C2ASHC
OPAQUE SETMCC
OPAQUE SETVCC
OPAQUE SETZCC
OPAQUE SIGNEX
OPAQUE CMNUL
OPAQUE C2MUL
OPAQUE CVNUL
OPAQUE CCNUL
OPAQUE CMDIV
OPAQUE C2DIV
OPAQUE CVDIV
OPAQUE CC DIV
ECHO
IV.III.4. Simulation Command Files: Unimplemented Instructions

1 UNIMPLEMENTED OPERATION BREAKS

DECHO
BREAK MFP  MFPI,MFPD
BREAK MTP  MTPI,MTPD
BREAK SPL
BREAK MUL
BREAK DIV
BREAK FPENT  CFCC,SETF,SETI,SETO,SETL
            CLRF,TSTF,ABSF,NEGOF,CLRDO,TSTDO,ABDO,NEGO
            DFPS,STFPS,STST
            MULF,MODF,ADDF,LDF,SUBF,CMPF,STF,DIVF,STEXP,
            STCFI,STCFL,STCFO,LOEXP,LOCIF,LOCFL,LOCOF,
            MULD,MODD,ADDD,LDD,SUBD,CMPD,STD,DIVD,STEXP,
            STCDI,STCDL,STCDF,LOEXPD,LOCID,LOCLD,LOCDF

ECHO