AN ABSTRACT OF THE THESIS OF

Faten A. Sakallah for the degree of Master of Science in Electrical and Computer Engineering presented on July 15, 1986.

Title: ISPS IN SYSTEM DESIGN

Redacted for Privacy

Abstract approved:

Roy C. Rathja

This dissertation demonstrates the utility of the Instruction Set Processor Specification (ISPS) in a design environment. Although ISPS supports a wide range of applications in the areas of architecture evaluation and certification, design automation, software generation and fault simulation, the emphasis in this dissertation is focussed on using it in design verification and performance evaluation of a 16-bit processor, AF85. The utility of ISPS to support fault simulation at the functional level is also illustrated by several examples applied on AF85.

ISPS IN SYSTEM DESIGN

by

Faten A. Sakallah

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed July 15, 1986

Commencement June 1987

APPROVED:

Redacted for Privacy

Associate Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented July 15, 1986

Typed by the author for Faten A. Sakallah

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ISPS IN SYSTEM DESIGN

I. INTRODUCTION

The evolution of VLSI into the world of digital system design has provided hardware engineers with a reason to develop a new approach for designing digital hardware. In the early days, the designer was able to keep track of the overall structure and behavior of a hardware design by using logic diagrams, boolean equations and the designer's own memory. As systems complexity increased in the VLSI era, it became more difficult for a designer to keep track of a design, resolve design problems, or even communicate the design with others. The need to describe the design in a higher order language was apparent. Hardware Description Languages (HDLs) evolved as a solution.

An HDL is similar to any other high level programming language, and provides a means of [32],[39],[42],

*precise yet concise description of design,

*convenient documentations to generate user's manuals,

*input of the design description into a computer for simulation and design verification at various levels of detail,

*software generation at the preprototype level, thus bridging the hardware/software development time gap,

*efficient incorporation of design changes and corresponding changes in documentation,

*designer/user (teacher/student) communication interface at the desired level of complexity.

A number of such HDLs have been developed, e.g. CDL, AHPL, ZUES, ISPS, etc., to describe digital systems and hardware algorithms.

A digital system can be described in the following six levels of complexity [42],[46],[47]. Different HDLs reflect different levels of abstractions of computer hardware.

1.Algorithmic level, which specifies only the algorithm used by the hardware for the problem solution.

2.Processor memory switch (PMS) level, which describes the system in terms of processing units, memory components, peripherals, and switching networks.

3.Programming level, where the instructions and their interpretation rules are specified.

4.Register transfer level, where the registers are system elements, and the data transfer between these registers are specifed according to some rule.

5.Switching circuit level, where the system structure consists of an interconnection of gates and flip-flops, and the behavior is given by a set of boolean equations.

6.Circuit level, where the gates and flip-flops are replaced by the circuit elements such as transistors, diodes,etc.

HDLs provide a means of attaining computer assistance in hardware design [32]. Thus the use of an HDL-based design approach is an efficient method for dealing with the increasing complexity of VLSI design projects. An HDL design approach is realized by implementing a HDL on a host computer, which then becomes an HDL-based design system.

ISPS, Instruction Set Processor Specification, is designed to describe precisely the programming level of a digital system [2,3,5,9],[43]. The behavior of the 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. Thus if we specify the nature of the operations and the rules of interpretations, the actual behavior of the processor depends on the initial conditions and the particular program.

ISPS supports a wide range of applications in the area of automatic design of both hardware and software [7]. These applications may be categorized in four major areas, namely

*evaluation and certification of instruction set processors

*design automation

*software generation

*functional fault simulation

The purpose of this dissertation is to demonstrate the advantages of using ISPS in hardware and software design, with emphasis on design verification, performance evaluation and functional fault simulation.

To achieve this goal, the dissertation presents an overview of ISPS and its applications, then uses the notation to aid the design process of a new 16-bit machine.

The project was undertaken with a secondary purpose in mind. This is to introduce and install a HDL at OSU to serve as a useful starting point in design automation.

Chapter II is devoted entirely to the review of a selected number of HDLs. This chapter provides information to help the reader familiarize himself with existing HDLs, their differences and similarites.

Chapter III presents an introduction of the ISPS notation, illustrated by examples on the PDP-8 computer system.

Chapter IV surveys the applications of ISPS in a design environment.

Chapter V presents the main features of a proposed architecture of a 16-bit processor. The complete hardware description of the design using ISPS is presented in Appendix A.

Chapter VI demostrates the utility of ISPS in design. The chapter introduces the use of the ISPS simulator in design verification and performance evaluation, by test programs, of the proposed architecture AF85. It concludes with demonstrating how to utilize the simulator for functional fault simulation. Complete listings of simulation runs are presented in Appendix B.

Finally, chapter VII gives a summary of the work presented in the paper.

II. FEATURE DESCRIPTION OF A NUMBER OF HDL's

2.0 Introduction:

Seven hardware description languaes are chosen for a feature study based on the following criteria [4],[32],[42],[46]

* environment in which each was created.
* end use environment.
* extent of use.
* spectrum of language features.
* currency.

The seven languages studied are:

1.Computer Design Language, CDL [15,16].

2.Interactive Design Language, IDL [28,29,30].

3.ZUES Hardware Description Language [18],[25].

4.A Hardware Programming Language, AHPL [20,21,22].

5.CONLAN, a CONsensus LANguage [36,37,38].

6.Very High Speed Integrated Circuits,VHSIC,hardware Description Language, VHDL [27],[28],[41].

7.Instruction Set Processor Specification, ISPS [2,3,4,5,7,9],[43].

Table 1 relates the chosen languages to the criteria listed.

In the following sections, a brief description of each language is presented. A set of language features is then defined, to provide a basis of comparison among the languages described. Finally, the seven languages are compared, according to these features, and the results of the comparison are summarized in Table 2.

Table 1

Selection Criteria for Described Languages

Criteria	IDL	CDL	AHPL	ZUES	CONLAN	VHDL	ISPS
environm. in which created	I	U	U	I	I/U	I	U
end use environm.	I	I/U	I/U	-	-	na	I/U
extent of use	R	A	A	R	-	na	W
spectrum of lang. features	L	L	L,S,H	L,S,H, C	L,S,H	L,S,H	L,H
currency	#10	#19	#6	#1	#1	#0	#8

Legend

Key	Explanation	Key	Explanation
I	industry	H	supports hierarchy description
U	university		
R	restricted use	C	supports circuit description
W	wide use in industry		
A	mainly academic	#	no.of years available to designers
L	supports logic\ function description	na	information not available
S	supports structure description		

2.1 CDL-Computer Description Language [15,16]:

CDL was first reported by Y. Chu in 1965 in the University of Maryland. Since then it has been used in teaching digital logic design. CDL describes the structural and functional parts of a digital system. Structural parts, such as memory, registers, clocks and switches, are declared explicitly at the beginning of the description. The functional behavior of the elements is described by the commonly used operators and user defined functions. Whenever there is a data transfer, a data path is declared implicitly. The organization of a CDL description includes a list of declaration statements, followed by a list of execution statements.

CDL allows synchronous timing mode only. Both parallel and sequential operations are allowed. CDL uses nonprocedural order of execution mechanism. That is, each CDL statement is associated with a label which describes the conditions under which execution is performed. All variables in a CDL description are global. The system description can be only at one level, i.e. there are no subroutine facilities, thus making CDL unsuitable for describing hardware in a modular fashion.

A translator and a simulator have been developed for CDL, both are implemented in FORTRAN. The translator performs a syntax check on the system description and then translates it into logic equations. The simulator executes the output of the translator by means of a set of simulation commands.

It is not possible to include special hardware components like integrated circuits in a CDL description. However, the FORTRAN implementation gives CDL simplicity of structure and portability which have contributed to its popularity.

2.2 IDL-Interactive Design Language [28,29,30]:

IDL is a hardware design language developed in early 1974 by L. Maissel and D. Ostapko at IBM. The language supports both structural and behavioral descriptions. The order in which IDL statements are listed is unimportant. Data entry may be in graphical flow chart or text form. IDL also allows blocks of logic to be represented as truth tables.

IDL is a nonprocedural language. It only allows synchronous timing mode. Sequence control is achieved through the use of labels. Every IDL statement, that is not a declaration, must be associated with a label. There is no restriction on how many labels may be active at any given time. If two or more labels are simultanously active, simulation treats them as parallel processes. The general action statement in IDL is IF-THEN-ELSE. Input conditions can be quite complex, and many complex functions such as rational operators are built in. Output statements can also be more complicated than just assigning values to outputs, for example, they can imply complicated control actions such as register transfers and memory accesses.

IDL is used for both design and description, as it generates two level logic from high level description. Multilevel logic can also be generated. Design verification under IDL is achieved by simulation.

IDL is mostly implemented in APL, although some CPU routines are implemented in IBM 370 assembly language.

IDL is used mostly in an industrial environment and it has found very limited use outside IBM.

2.3 ZUES-A Hardware Description Language [18],[25]:

ZUES is a general hardware description language developed in early 1980 at GTE Laboratories. ZUES supports both functional and structural descriptions. The functional description is usually linked to the structural description by giving each structural element a functional meaning. Hierarchical abilities are provided through the so called COMPONENT- TYPE within a description.

ZUES can generate hardware, conditionally, through the use of a simple metalanguage. However all feedback loops must lead through registers. This restriction is claimed to prevent poor design approaches and to simplify simulation.

ZUES is a nonprocedural language. Conditional statements such as IF statements and replication statements such as FOR statements are used to specify hardware.

ZUES is so close to the hardware level and offers good features to describe layout information that it is possible to write an efficient silicon compiler that produces reasonably efficient chips in terms of area, time and power.

Time in ZUES is assumed to proceed in discrete steps called clock cycles. Clock cycles are assumed to be long enough to allow signals sufficient time to ripple through the combinational logic.

2.4 AHPL-A Hardware Programming Language [20,21,22]:

AHPL is a hardware description language based on the APL notation. The language was designed by Hill and Peterson in late 1960's at the University of Arizona. The language has been used, since then, to teach digital system design. AHPL is a procedural, single-block language which supports both structural and functional descriptions. AHPL statements are normally executed in the order listed, unless a branch command changes the sequence.

AHPL makes use of only those APL operators which can be interpreted as hardware primitives. A few more have been added to AHPL representing unique hardware capabilities such as parallel control sequences, asynchronous transfers and conditional transfers. AHPL operators are selected from APL operators based on the set of hardware primitives to be used in a design, hence, the set of AHPL operators may differ from design to design. Declarations are very rare in AHPL descriptions.

AHPL descriptions may be synchronous or asynchronous. A compiler and a simulator have been developed for AHPL, both are implemented in FORTRAN.

2.5 CONLAN-A Consensus Language [36,37,38]:

CONLAN is a consensus hardware description language. The CONLAN project began in 1973 by establishing the CONLAN group, consisting of R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill and P. Skelly. The groups' main objective was to consolidate existing HDL's into a standard language.

The CONLAN approach utilizes a family of related languages rather than a single consensus language. These languages are linked by a common syntax and semantics. Each new language definition is based upon an existing one, and can be derived when the need arises.

The CONLAN group prepared the root language called Base ConLan, BCL.The language provides the basic object types and operations to describe the behavioral and structural parts of a digital system in both space and time.

2.6 VHDL-Hardware Description Language [27],[28],[41]:

VHDL is a new language developed to support the Department of Defense Very High Speed Integrated Circuit (VHSIC) program. The VHDL project began in 1983 by a team from Intermetrics, IBM and Texas Instruments.

VHDL provides a standard textual means of describing hardware components at levels ranging from the logic gate level to the digital system level.

The language organization provides a hierarchical design capability that allows the designer to describe, evaluate and utilize design alternatives. The key element within VHDL is the design entity. Each design entity has an interface and a body. The interface description represents the intended external interface of the hardware being designed. As design progresses, the interface description will be refined to match the real hardware exactly. Within the design body, the designer may describe one or more design alternatives, variants, for the desired hardware. Within each variant, several design aspects may be described in terms of function, RT level or pure structure.

A simulation model may be created, in the simplest case, by selecting a design entity and choosing the desired design alternative from that design entity. If structural description is chosen, a model tree may be created by looking at the structure contained within the chosen design entity and at each lower level design entity that is part of the structure. This process continues until all the design entities in lower level structures, for which functional or RT level descriptions exist, have been chosen.

The primary use of VHDL is in the design of VHSIC class of components. The designer will use the VHDL design system to specify designs, to create new design entities and to utilize existing design entities.

2.7 ISPS-Instruction Set Processor Specification [2,3,4,5,7,9],[43]:

ISPS is the second implementation of the ISP notation first introduced in Bell and Newell in 1971 [12]. ISPS was developed as a computer language by M. Barbacci at Carnegie-Mellon University in 1977. Although ISPS may be viewed as a high level programming language, its notation was developed to describe computers and other digital systems. ISPS is a register transfer language designed to support a wide range of applications rather than a wide range of levels.

The main purpose of the language is the description of the behavior of an instruction set processor, its data types and operations. Primitive data types include, registers, memories and transfer paths. Primitive operations include logic (single and multibit), arithmetic (several representations), and control (sequential, parallel, conditional).

ISPS is a block structured, procedural language capable of describing both synchronous and asynchronous designs. A compiler and a simulator have been developed for the language. Both are implemented in BLISS-10, thus limiting the portability of the software.

2.8 Features of Comparison:

The previous sections show that different languages address different features. Some languages provide more suitable constructs than others for the specification of the different features. By analyzing these languages, four main features that appeared to be useful as a basis of comparison were identified [32],[46]. These features are:

1.Levels of description.

2.Type of description.

3.Sequencing mechanism.

4.Timing modes.

Table 2 summarizes these features for the languages reviewed.

1.<u>Level of description:</u>

Tools for describing digital hardware at the circuit level and the switching circuit level, have existed for years. HDL's raised the level of abstraction to the register transfer level, and the trend is to expand the capability of HDL's to multi and mixed levels of description. There are five discrete levels at which a HDL can be used [39],[42],[46]. These are,

a.PMS level or system level: The top level of description, evaluates the gross properties of the computer system. Its elements are processors, memories, switches, peripheral units,etc. and the parameters are costs, information flow rates, power,etc.

b.Programming level: The basic components are the interpretation cycle, the machine instructions and operations. The behavior of the 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. Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends only on the initial conditions and the particular program.

c.Register transfer level or functional level: Data flow and control operate in discrete steps. A combination of switching circuits is used to form registers, register transfers and other data operations. The elements (registers) are combined (transformed) according to some rule and then stored (transferred) into another register. The rules of transformation can be almost anything, from simple transfers to complex logical and arithmetic expressions.

d.Switching circuit level: The system structure is given by a collection of gates and flip-flops, and the behavior by a set of boolean equations. Timing is carried out at a finer degree than at the preceding level, a time unit being usually on the order of a gate delay.

e.Circuit level: Gates are described as some interconnection of diodes, transistors, resistors,etc.

according to electrical circuit laws. Most of the discrete properties of the previous two levels are lost, and timing is carried out at a finer degree, where transient behavior is an important consideration.

2.Type of Description:

Three discrete levels of detail can be used to describe a digital system, structural, functional and behavioral [32],[46],[47].

Structural description represents a system in terms of the actual hardware components and their interconnections. A functional description suppresses low level structural details, so that a designer deals with registers and logic networks rather than individual flip-flops and logic gates. The action of a system is described as an algorithm involving these higher level components. A behavioral description offers an even higher level of abstraction. This type of description is concerned with the order in which operations take place, but not necessarily with the values of the data that is being manipulated by the digital system. This type of description is very close to conventional programs in most programming languages.

3.Sequencing Mechanism:

A hardware description language can be classified as a procedural or a nonprocedural language [42]. In a procedural language, statements are executed in the order in which they are written, unless a specific control transfer statement is used. In a nonprocedural language, all statements are considered capable of being active in any order. Some sort of control expression is associated with each statement, to indicate under what condition the statement is to be executed.

4.Timing Modes:

This feature refers to the ability of a HDL to describe a system as synchronous, asynchronous or a mixture of both [46].

Among all the languages reviewed. ISPS was the only language that had a supporting software package which can be installed on the DEC-20 computer system. The software package was provided by M. Barbacci, Department of Computer Science, Carnegie-Mellon University. In addition to that, ISPS had the greatest share of publications related to the language and its applications.

Table 2

Comparison of Language Features

Feature Supported	IDL	CDL	AHPL	ZUES	CONLAN	VHDL	ISPS
level of Descrip.							
PMS	*		*	*	*	*	*
Programming							*
Register Transfer		*	*				*
Switching Circuit	*	*	*	*	*	*	*
Circuit			*				
Type of Descrip.							
Functional	*	*	*	*	*	*	*
Behavioral	*		*		*		*
Structural	*	*	*	*	*	*	
Sequencing Mechan.							
Procedural			*				*
Nonprocedural	*	*		*			
Timing Modes							
Synchronous	*	*	*	*	*	*	*
Asynchronous			*		*	*	*
Mixed Mode						*	

III. INTRODUCTION OF ISPS

3.0 Introduction:

This chapter presents an overview of the ISPS notation. The information presented in this chapter is essentially a summary of "An ISPS Primer for the Instruction Set Processor Notation" by M. Barbacci [11]. The examples given are meant to cover enough of the language to provide a "reading" capability. Thus, while this overview, in itself, might not be sufficient to allow writing ISPS descriptions, it should be detailed enough to permit the reading and study of complex descriptions.

3.1 Instruction Set Processor Description:

ISPS is a computer hardware description language, based on the notation of ISP, first introduced by Bell and Newell in 1971. To describe the ISP of a computer, or any machine, we need to define the operations, instructions, data types, and interpretation rules used in the machine. These will be introduced, gradually, as the primary memory state, processor state, and the interpretation cycle are described. Primary memory is not, strictly speaking, part of the instruction set processor, but it plays such an important role in its operation, that it is usually included in the description. Data types, such as integers, floating point numbers, characters etc., are, in general, abstractions of the contents of the machine registers and memories. One data type that requires explicit treatment is the instruction, which will be discussed later in greater detail.

The ISPS description of the PDP-8 will be used as a source of examples. In the presentation of the PDP-8 registers and data types, the following conventions will be used:

1.Names in uppercase correspond to physical components of the PDP-8, PROGRAM COUNTER, INTERRUPT LINES, etc.

2.Names in lowercase do not have a corresponding physical component, instruction mnemonics, instruction fields, etc.

3.1.1 Memory State:

The description of the PDP-8 begins by specifying the primary memory used to store data and instructions.

```
M\Memory[0:4095]<0:11>,
```

The primary memory is declared as an array of 4096 words, each 12 bits wide. The memory has a name M and an alias Memory. These aliases are a special kind of comments, and are useful for indicating the meaning or usage of a register's name. ISPS identifiers, as in most programming languages, consist of letters and digits, beginning with a letter. The character "." is also allowed to increase readability.

The expression [0:4095] describes the structure of the array. It declares the size, 4096 words, and the names of the words,0,1,...,4094,4095. In a similar manner, the expression <0:11> describes the structure of each word. It declares the size, 12 bits, and the names of the bits,0,1,...,10, 11.

In the PDP-8, memory is divided into 128-word pages. Page zero is used for holding global variables and can be accessed directly by each instruction. Locations 8 to 15 of page zero have the special feature, auto indexing, such that when accessed indirectly the contents of the location is incremented by 1. These memory regions can be described as part of M as follows:

```
P.O\Page.Zero[0:127]<0:11> :=M[0:127]<0:11>,
A.I\Auto.Index[0:7]<0:11> :=P.O[8:15]<0:11>,
```

Note that A.I[0] corresponds to P.O[8], A.I[1] corresponds to P.O[9] and so on.

3.1.2 Processor State:

The processor state is defined by a collection of registers used to store data, instructions, condition codes, etc. during the instruction interpretation cycle.

The PDP-8 has a 1-bit register L, which contains the overflow or carry generated by arithmetic operations, and a 12-bit register AC, which contains the results of arithmetic and logic operations. The concatenation of L and AC constitutes an extended accumulator LAC. The structure of LAC is described below,

```
LAC<0:12>,
          L\Link<> ::=LAC<0>,
          AC\Accumulator<0:11> ::=LAC<1:12>,
```

The expression <> indicates a single unnamed bit. The Program Counter is used to store the address of the current instruction being executed,

```
PC\Program.Counter<0:11>,
```

In the PDP-8, I\O devices are allowed to interrupt the central processor by setting the INTERRUPT.REQUEST flag. The processor can honor these requests or not depending on the setting of the INTERRUPT.ENABLE bit. These are described as follows:

```
INTERRUPT.ENABLE<>,
INTERRUPT.REQUEST<>,
```

There are 12 console switches which can be read by the processor. These are treated as a 12-bit register,

```
          SWITCHES<0:11>,
```

3.1.3 Instruction Format:

In PDP-8, instructions are 12-bits long, each contains an operation code and an operand address. This structrue is described as follows:

```
        i\instruction<0:11>,
                    op\operation.code<0:2>:=i<0:2>,
                    ib\indirect.bit<>:=i<3>,
                    pb\page.0.bit<>:=i<4>,
                    pa\page.address<0:6>:=i<5:11>,
```

op, ib, pb, and pa are abstractions that allow us to treat selected fields of the PDP-8 instruction as individual entities.

3.1.4 Partioning the Description:

In ISPS, a description may be divided into sections of the form:

```
            **section.name**
             <declaration>,
             <declaration>,
            ...............
            **section.name**
             <declaration>,
             <declaration>,
             ................
```

Each section begins with a header, an identifier enclosed between ** and **. A section consists of a list of declarations separated by commas. Section names are not reserved key words in the language. The register and memory declarations presented earlier can be grouped in the following sections.

```
** Memory.State **
M\Memory[0:4095]<0:11>,
     P.0\Page.zero[0:127]<0:11>  :=M[0:127]<0:11>,
     A.I\Auto.Index[0:7]<0:11>   :=P.0[8:15]<0:11>,
** Processor.State **
LAC<0:12>,
     L\Link<>                :=LAC<0>,
     AC\Accumulator<0:11> :=LAC<1:12>,
PC\Program.Counter<0:11>,
RUN<>,
INTERRUPT.ENABLE<>,
INTERRUPT>REQUEST<>,
SWITCHES<0:11>,
** Instruction.Format **
i\instruction<0:11>,
     op\operation.code<0:2>  :=i<0:2>,
     ib\indirect.bit<>       :=i<3>,
     pb\page.0.bit<>         :=i<4>,
     pa\page.address<0:6>    :=i<5:11>,
     IO.SELECT<0:5>          :=i<3:8>,
          .
          .
          .
```

Note that one more field was added,IO.SELECT, and its declaration is associated with a preassigned portion of register i. This feature adds flexibility to the ISPS description. A comment is indicated by "!", and all characters following it to the end of the line are treated as commentary.

3.2 Effective Address:

The effective address computation is an algorithm that computes addresses of data and instructions. The description of the algorithm follows,

```
** Effective.Address **
last.pc<0:11>,
eadd\effective.address<0:11>  :=
     Begin
     Decode pb=>
        Begin
        0:= eadd ='00000 @ pa,       !Page Zero
        1:= eadd = last.pc<0:4> @ pa !Current Page
        End Next
     If Not ib => Leave eadd Next
     If eadd<0:8> Eqv  001 =>
        M[eadd] = M[eadd] + 1 Next   !Auto Index
     eadd = M[eadd]
         End,
```

As was mentioned before, the instruction reserves 9-bits for addressing information. These bits, together with some other portions of the processor state, are interpreted by the algorithm to yield the necessary 12-bits of addressing needed for the 4096 words memory.

3.2.1 Address Computation:

In the PDP-8, the concept of locality of memory reference is used to reduce the addressing information by assuming that data are usually in the same page as the instructions that reference them. The pa field of an instruction means address within current page. The pb field is used to indicate when pa is to be used as an address within page 0 instead of current page. last.pc contains the address of the current instruction and is used to compute the current page number.

The first step of the algorithm,

```
    Decode pb=>
            Begin
            0:= eadd = '00000 @ pa,
            1:= eadd = last.pc<0:4> @ pa
            End Next
```

indicates a number of alternative actions, to be selected according to the value of the expression following the Decode operator. The alternatives appear enclosed between Begin and End and separated by ",". The expressions "0:=" and "1:=" are used to label the statements with the corresponding value of pb. If the alternative statements are left unnumbered, they will be treated as if they were labelled "0:=", "1:=", "2:=", etc.

The effective address, eadd, is built by concatenating, @, the page number with the page address, pa. If pb=0, then page 0 is used in the computation. If pb=1, then bits 0 through 4 of last.pc, which indicate current page, are used in the computation. Note that the 5-bits of page number together with the 7-bits of page address constitute the 12-bits needed for addressing. Note also, that constants prefixed with the character "'" represent binary numbers.

3.2.2 Indirect Addresses:

In the PDP-8, indirect addresses are specified by a bit in the instruction register, ib. The second step of the algorithm,

```
If Not ib=> Leave eadd
```

is separated from the previous step by the Next operator. Statements preceding Next must be completed before the statements following it can be executed.

The first step of the algorithm computed a preliminary eadd. The second step tests the value of ib, if it is 0, direct addressing, then the preliminary eadd is used as the real eadd. If ib is equal to 1(indirect addressing) then the computed eadd is used to access a memory location that contains the real eadd. In the former case, the expression Leave eadd, is similar to a RETURN statement in many programming languages.

3.2.3 Auto Indexing:

Constants prefixed with the character " " represent octal numbers. The third step of the algorithm,

```
If eadd<0:8> Eqv 001 => M[eadd] + 1 Next
eadd = M[eadd]
```

compares the high order bits of eadd with 001. If they are equivalent, the memory location is first incremented and the new value is used as the indirect address. Now regardless of whether auto indexing took place or not, the last step of the algorithm uses the preliminary effective address, which could have been modified by auto indexing, as the address of a memory location which contains the real effective address.

3.3 Instruction Interpretation:

The instruction interpretation section describes the instruction cycle, i.e. the fetching, decoding and executing of instructions.

```
** Instruction.Interpretation **
interpret :=
     Begin
     Repeat  Begin
        i = M[PC]; last.pc = PC Next
        PC = PC + 1 Next
        execute() Next
```

```
                If INTERRUPT.ENABLE And INTERRUPT.REQUEST
                    Begin
                    M[0] = PC Next
                    PC = 1
                    End
                  End
            End,
```

The instruction cycle is described by a loop. The "Repeat" operator precedes a block of statements that are to be continuously executed. The instruction cycle of the machine consists of four steps:

1.A new instruction is fetched, i = M[PC].

2.The program counter is incremented, PC = PC + 1.

3.The instruction is executed, execute().

4.Interrupt requests, if allowed are honored. The cycle is then repeated.

The ";" separator is used to indicate concurrency. The execute procedure describes the individual instructions.

```
        execute :=
           Begin
           Decode op =>
             Begin
             #0\and := AC = AC And M[eadd()],
             #1\tad := LAC = LAC + M[eadd()],
    #2\isz := Begin
                   M[eadd] = M[eadd()] + 1 Next
                   If M[eadd] Eql 0 => PC = PC + 1
                   End,
         .
         .
         .
       End
    End,
```

From above, notice the different uses of eadd in the statement, M[eadd] = M[eadd()] + 1. The effective address is computed once, eadd(), and is then used to fetch the memory location, M[eadd()]. The result of the addition must be stored back in the same memory location. This is indicated by using the effective address register, eadd, on the left hand side, M[eadd]. eadd already contained the correct address and there was no need to compute it.

3.4 Other Features of ISPS:

Not all the features of ISPS have been presented in the previous examples. This section provides a list of the missing operations.

1.Constants:

A constant is a sequence of characters drawn from some alphabet, determined by the base of the constant. The alphabets for the predefined bases in ISPS are:

Base	Prefix	Alphabet
2	'	0,1,?
8	#	0,1,2,3,4,5,6,7,?
10		0,1,2,3,4,5,6,7,8,9,?
16	"	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,?

The character "?" can be used to specify a don't care digit in the corresponding alphabet.

The length of a constant is measured in bits. Decimal constants are one bit longer than the smallest number of bits needed to represent its value (see Table 3).

Table 3

Representation of Constants in ISPS

Example	Length	Bit Pattern
"1000	16	0001000000000000
15	5	01111
#17	6	001111
0	2	00
'0?101	5	0?101
#?2	6	???010

2.Arithmetic Representation:

ISPS allows the user to specify arithmetic operations in four different representations using the following modifiers:

Modifier	Arithmetic Representation
{TC}	Two's Complement
{OC}	One's Complement
{SM}	Sign Magnitude
{US}	Unsigned Magnitude

The above modifiers can be attached to any arithmetic or relational operator to override the default, two's complement. They can also be attached to a procedure declaration or a section name.

```
      test :=
          Begin {OC}                   !Default for the body
           ...
          End,


     ** Section.1 ** {TC}              !Default for the section
        ...........


       X = Y + {SM} Z                   !Instance
```

3.Sign Extension:

All ISPS data operators define results whose length is determined by both the lengths of the operands and the specific operator. Some operations,however require that their operands be of the same length. This is usually done by "sign-extending" the operands. For Unsigned Magnitude arithmetic, "sign- extension" is interpreted as zero extension on the left. In One's and Two's complement arithmetic, the expansion is done by replication of the sign bit. In Sign Magnitude arithmetic, the expansion is done by inserting zeros between the sign bit and the most significant bit of the operand.

3.5 Data operators:

3.5.1 **Negation and Complement:-,** NOT

Negation generates the arithmetic complement of the operand, the result is one bit longer than the operand. Not generates the logical complement of the operand, the result is the same length as the operand.

3.5.2 **Concatenation:**

The @ operator concatenates the two operands, the length of the result is the sum of the lengths of the operands.

3.5.3 **Shift and Rotate:**SL0, SL1, SLD, SLR, SR0, SR1, SRD, SRR

These operators shift or rotate the left operand the number of times indicated by the right operand. The result has the same length as the left operand. The operators have the format "Sxy", where "x" is either Left or Right, "y" is either 0, 1, Duplicate or Rotate. Sx1, shifts the left operand by inserting 1's in the vacant positions. Sx0 inserts 0's. SxD inserts copies of the bit leaving the position to be vacated. SxR inserts copies of the bit being shifted out.

3.5.4 Addition and Subtraction: +, -

These operators compute the arithmetic sum and difference of the two operands. The shortest operand is sign-extended, and the result is one bit longer than the largest operand.

3.5.5 Multiplication, Division and Remainder: *, /,MOD

These operators compute the arithmetic product, quotient, and remainder of the two operands. The lengths of the results are:

Operation	Length of Result
*	sum of lengths
/	left operand, dividend
MOD	right operand, divisor

3.5.6 Relational Operations: EQL, NEQ, LSS, LEQ, GTR, GEQ, TST

These operations perform arithmetic comparison between the two operands. The shortest operand is sign extended, the result is either one or two bits long.

3.5.7 Conjunction and Equivalence: AND, EQV

These operators produce the logical product and coincidence of the two operands. The shortest operand is zero extended, and the result is as long as the largest operand.

3.5.8 Disjunction and Non-equivalence: OR, XOR

These operators produce the logical sum and difference of the two operands. The shortest operand is zero extended, and the result is as long as the largest operand.

3.5.9 Logical and Arithmetic Assignment: =, <=

The logical assignment operator, "=", zero extends the source to match the length of the destination. The arithmetic assignment operator, "<=", sign extends the source to match the length of the destination.

3.6 Control Operators:

The LEAVE, TERMINATE, RESTART, and RESUME operators are used to terminate the execution of an action. The LEAVE operator is used to force the termination of an action. This operation must be enclosed inside the action being terminated. TERMINATE, is essentially equivalent to LEAVE but is not required to be enclosed within the action. The RESTART operator is used to abort the current execution of a procedure and then reinitiates it. The RESUME operator is similar to the LEAVE operator, if we think of LEAVE as a "return from", and RESUME as a "return to".

3.7 Predeclared Procedures:

The following procedures are predeclared in the ISPS notation.

COUNT.ONE(expression)<..>: when activated, it returns the number of non zero bits in the expression. The length of the result is equal to the decimal value of the length of the expression.

DELAY(expression): when used, does not have side effects. DELAY ends its activation after a number of application-defined time units given by expression.

FIRST.ONE(expression)<..>: when activated, returns the number of leading zeros in the value of the expression. The length of the result follows the rules defined for COUNT.ONE.

IS.RUNNING(procedure)<..>: when used, returns 1 if procedure is currently active, 0 otherwise.

LAST.ONE(expression)<..>: when activated, returns the number of trailing zeros in the value of the expression. The length of the result is identical to that of COUNT.ONE.

MASK.LEFT(expr1,expr2)<..>: when activated, returns a result of the same length as expr1. The leading expr2 bits are set to 0, the remaining bits retain the value they had in expr1.

MASK.RIGHT(expr1,expr2)<..>: when activated, identical to MASK.LEFT, but it clears the bits on the right of expr1 using expr2 to compute the number of bits.

NO.OP() : when activated has no side effects, can be used as a null action.

PARITY(expression)<>: when activated, returns the odd parity bit of the expression.

STOP(): when used, terminates the activation of all procedures.

TIME.WAIT(expr1,expr2)<..>: when activated, it computes expr2 once, then continuously evaluates expr1 until it is non zero or the number of time units represented by expr2 is exceeded. At the end, it returns the final value of expr1. Depending on this value, the caller can decide whether expr1 yielded a non zero value, or the time out limit given by expr2 was exceeded before expr1 became non zero.

UNDEFINED()<..>: when activated, returns a carrier of undetermined length, and whose value is unknown. The activation of UNDEFINED is terminated after some undetermined amount of time.

UNPREDICTABLE(): when activated, it is not guaranteed to terminate, or upon terminaion, control will not return to activation site.

<u>WAIT(expression)<..>:</u> when activated, it continuuously evaluates the expression. Its action is terminated when the value of the expression is not equal to 0. The result of WAIT is the last value of the expression.

IV. APPLICATIONS OF ISPS

4.0 Introduction:

The goal of designing ISPS was to develop a computer description language that would be appropriate for diverse applications. This chapter presents an overview of the applications of ISPS in a design environment [7]. These include Evaluation and Certification of Instruction Set Processor, Design Automation, Software Generation, and Functional Fault Simulation. The information described is meant to introduce several ideas of how to utilize the language in research at OSU. Some of these ideas are examined in detail and applied to the design of a new 16-bit architecture in a later chapter.

4.1 Evaluation and Certification of Instruction Set Processor:

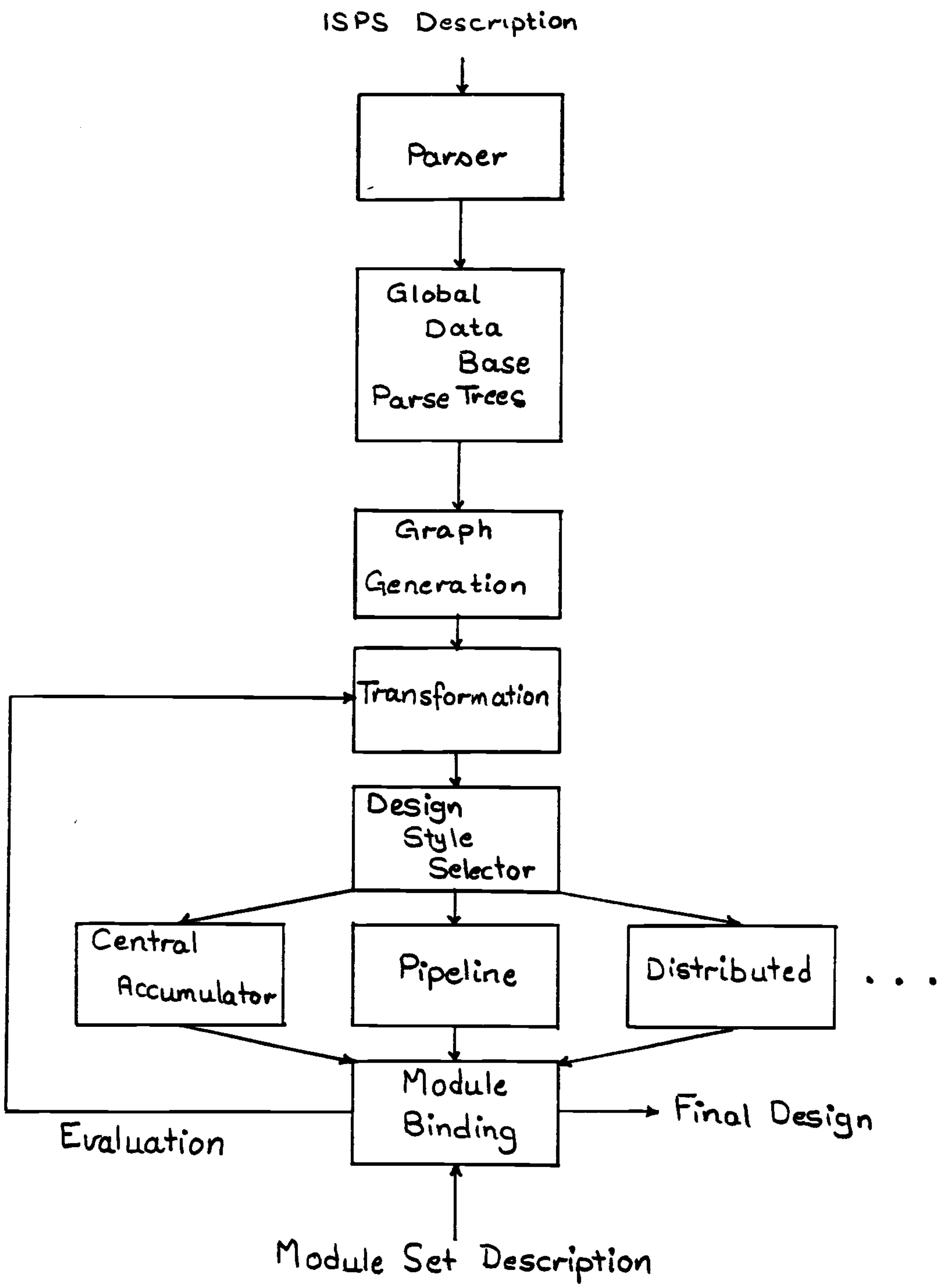
A new approach was developed to select a computer architecture for certain applications [10],[44]. This approach departed from the traditional measure, used from typical computer performance studies, namely the execution speed of a test program [17]. A new set of alternative measures were defined. These include, S, the number of bytes representing a test program as a space measure. The execution time measures include M, the amount of information transferred between primary memory and the processor, and R, the amount of information transferred inside the processor during the execution of a test program.

The architecture evaluation is based on the computation of the S, M, and R measures for a collection of test programs [10],[13]. Using the simulator facility of ISPS, one could also verify the correctness of the machine descriptions by running the manufacturer's machine diagnostics on the simulated machine.

4.2 Design Automation:

ISPS has been used extensively at Carnegie-Mellon University (CMU) in design automation. The CMU Register Transfer Computer Aided Design (RT-CAD) system [7],[35],[45] is shown in Figure 1. The system accepts the ISPS description of a target machine as one of its inputs and the description of physical components as the other. The specification of the target machine is first translated into a graph representation of behavior. By using a set of graph transformation algorithms, this initial graph may be transformed into alternative graphs, all of which represent the same behavior.

Given two graphs and a set of user goals, the design system can automatically accept or reject an alternative design. In the design style selector phase, a design style is selected according to a set of rules that guide the interconnections of the abstract components used in the behavioral graph representation.



The CMU RT-CAD System

Figure 1

Once a design style is selected, a design style allocator is used to generate a layout of the registers, functional units, data paths and their interconnections. The module binding phase uses the information gathered in the previous phase as well as information stored in the module data base, to select the physical components and the order in which they are bound to the abstract components specified in the layout. The output of the module binding phase is then evaluated, and the result is reported back to the previous stages.

Once a design is selected as the best of the alternatives considered, the module binding phase generates the necessary information needed by the physical design system. This is the place where a traditional design automation system starts, i.e. with logic diagrams in which components and interconnections are completely specified.

4.3 Software Generation:

A current research topic is the automatic compiler-writing systems, called the Production Quality Compiler Compiler (PQCC) [7]. These systems produce compilers that are competative with hand generated compilers in every respect. To achieve this goal, the PQCC system must operate from descriptions of both the source language and the target computer. For more information on this topic consult B. Leverett, Department of Computer Science, CMU.

4.4 Functional Fault Simulation:

The ISPS simulator allows the specification of a variety of data and control faults at the functional level [6,7],[34]. The faults which can be simulated include hard and transient, occurring deterministically or probabilistically, stuck-at and shorted, data, control and operation types.

Each fault injected at the ISP level can provide the coverage of multiple faults at lower levels. For instance, consider the effect of the following fault:"the sign bit of R2 is always 0". At the ISP level, the fault is clearly defined and its effect, force all data from R2 to be positive, can be easily traced. By running a test program with diagnostic capability, the identification of the faulty function can be used to narrow down the identification of the physical fault. Given the ever decreasing costs of hardware, this type of diagnostic ability might be enough to allow the replacement of the faulty board.

V. ISPS MODEL OF A 16-BIT MACHINE

5.0 Introduction:

AF85 is a 16-bit machine designed with TTL parts. From the user's point of view, the machine supports the basic arithmetic-logic instructions, subroutine facilities, data manipulation instructions, such as SHIFT, and a powerful branching structure. The machine also provides the user with the ability to use a number of different addressing modes.

The following sections are essentially a summary of the AF85 system description presented in reference [1].

5.1 Main Features of AF85:

AF85 is a 16-bit machine with a 16-bit PC, 16-bit SP and five 16-bit general purpose registers. All data, control and address transfer in AF85 is accomplished via 16-bit single bus system, which results in a word addressable memory M[0:32767]. A relatively horizontal microprogrammed control unit is implemented for AF85.

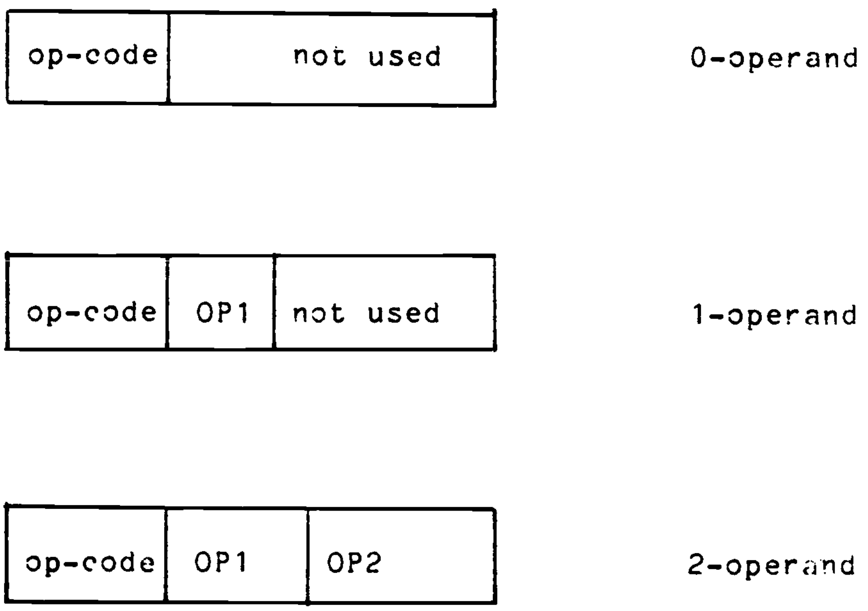
5.2 System Description:

5.2.1 Processor State: The central processor of AF85 contains the control logic and data paths for instruction fetching and execution. Processor instruction act upon operands located either in memory or in one of the five general purpose registers. These operands are 16-bit words.

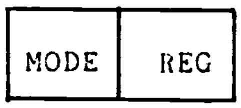
The general registers are 16-bits in length, and are referred to as R0 through R5. R4 is used as the PC and R5 as the SP.

Data manipulation instructions fall into two categories: arithmetic instructions, which interpret their operands as 2's complement integers, and logic instructions, which interpret their operands as bit vectors. A set of condition code flags (CC) is implemented by the processor, and is updated according to the sign and presence of carry/overflow from the result of any data manipulation instruction. The condition codes are contained in a processor status word (PSW).

5.2.2 Instruction Set & Addressing Modes: AF85 supports three types of instructions, namely the 0-operand, 1-operand and 2-operand instructions. The instruction formats are:



where each operand field has the following format:



Mode is a 2-bit field to specify the addressing mode, REG is a 3-bit field to specify the register indentification number.

Addressing Modes: There are four addressing modes which are coded in two bits,

00	immediate	OP <- M[PC]
01	register	OP <- [REG]
10	memory	OP <- M[M[PC] + [REG]]
11	indirect	OP <- M[M[PC]]

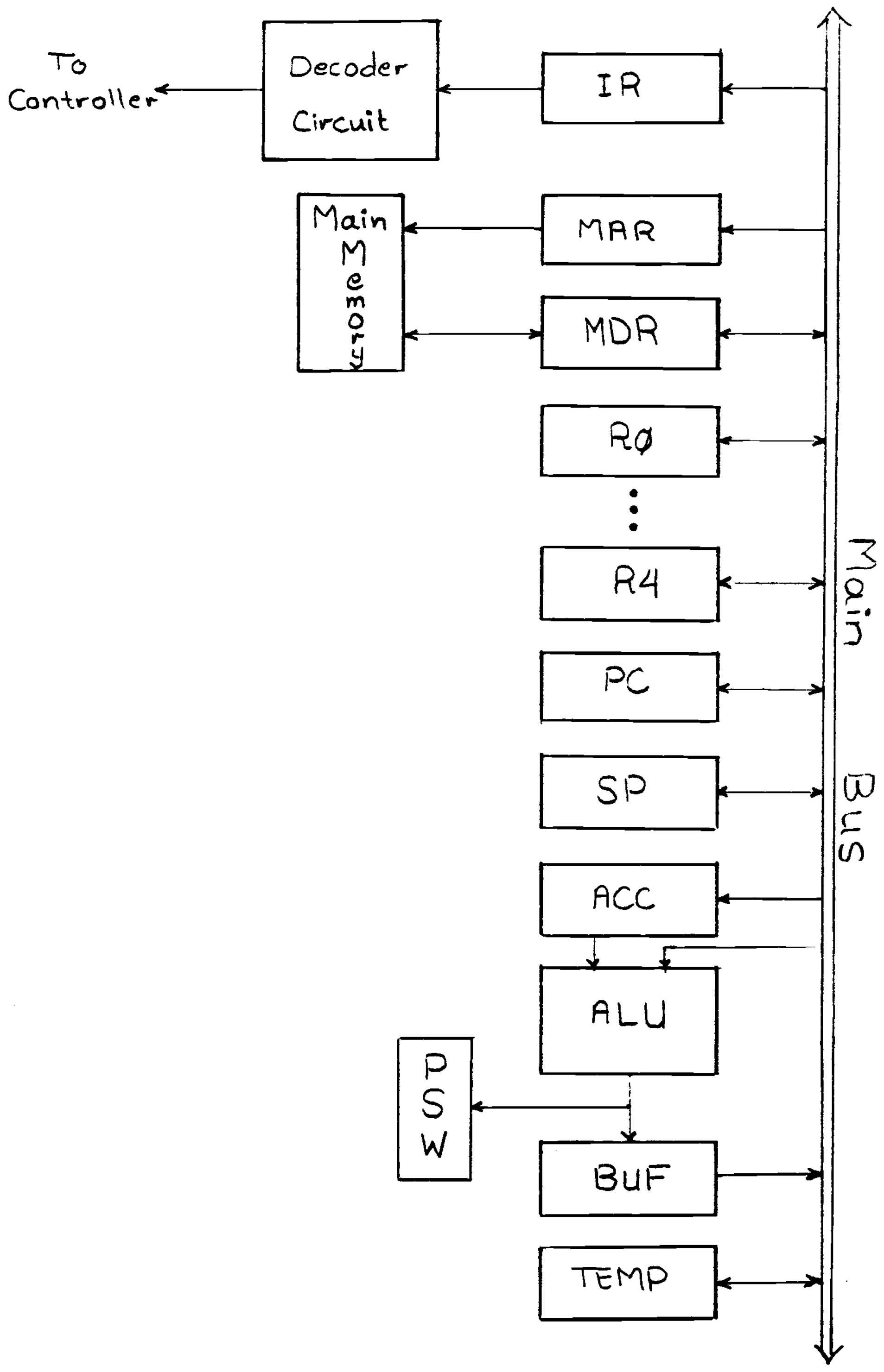
Instruction Set:

OPCode	Instruction	
0	ADD	OP2 <- OP1 + OP2 S.CC.
1	SUB	OP2 <- OP1 - OP2 S.CC.
2	CMP	OP2 - OP1 S.CC.
3	AND	OP2 <- OP1 AND OP2 S.CC.
4	OR	OP2 <- OP1 OR OP2
5	XOR	OP2 <- OP1 XOR OP2
8	SHL	if OP1>0, shift OP2 left OP1 times, S.CC.
7	SHR	if OP1>0, shift OP2 right OP1 times, S.CC.
6	MOV	OP2 <- OP1
17	RET	PC <- M[SP], SP <- SP + 1
35	CALL	SP <- SP - 1, M[SP] <- PC, PC <- OP1

33	PUSH	SP <- SP - 1, M[SP] <- OP1
34	POP	OP1 <- M[SP], SP <- SP + 1
18	STKADD	[TOS-1] <- [TOS] + [TOS-1], SP <- SP-1 S.CC.
19	STKSUB	[TOS-1] <- [TOS] - [TOS-1], SP <- SP-1 S.CC.
20	STKOR	[TOS-1] <- [TOS] OR [TOS-1], SP <- SP-1
21	STKAND	[TOS-1] <- [TOS] AND [TOS-1], SP <- SP-1
22	STKXOR	[TOS-1] <- [TOS] XOR [TOS-1], SP <- SP-1
23	STKCMP	[TOS] - [TOS-1], S.CC.
24	HLT	
32	CLR	OP1 <- 0
48	BR	PC <- OP1
49	BEQ	Z=1, PC <- OP1
50	BMI	N=1, PC <- OP1
51	BCS	C=1, PC <- OP1
52	BVS	V=1, PC <- OP1
53	BLT	(N .XOR. V)=1, PC <- OP1
54	BLE	Z .OR. (N .XOR. V)=1, PC <- OP1
55	BLO	(C + Z)=1, PC <- OP1
16	NOP	NO OPERATION
56	BNE	Z=0, PC <- OP1
57	BP	N=0, PC <- OP1

58	BNC	C=0, PC <- OP1
59	BNV	V=0, PC <- OP1

5.2.3 Data Flow: All registers, ALU, and memory are connected via a single bi-directional bus. The following briefly describes the various components on the data flow diagram given in Figure 2, and the different ways these components are connected.



Data Flow Diagram of AF85

Figure 2

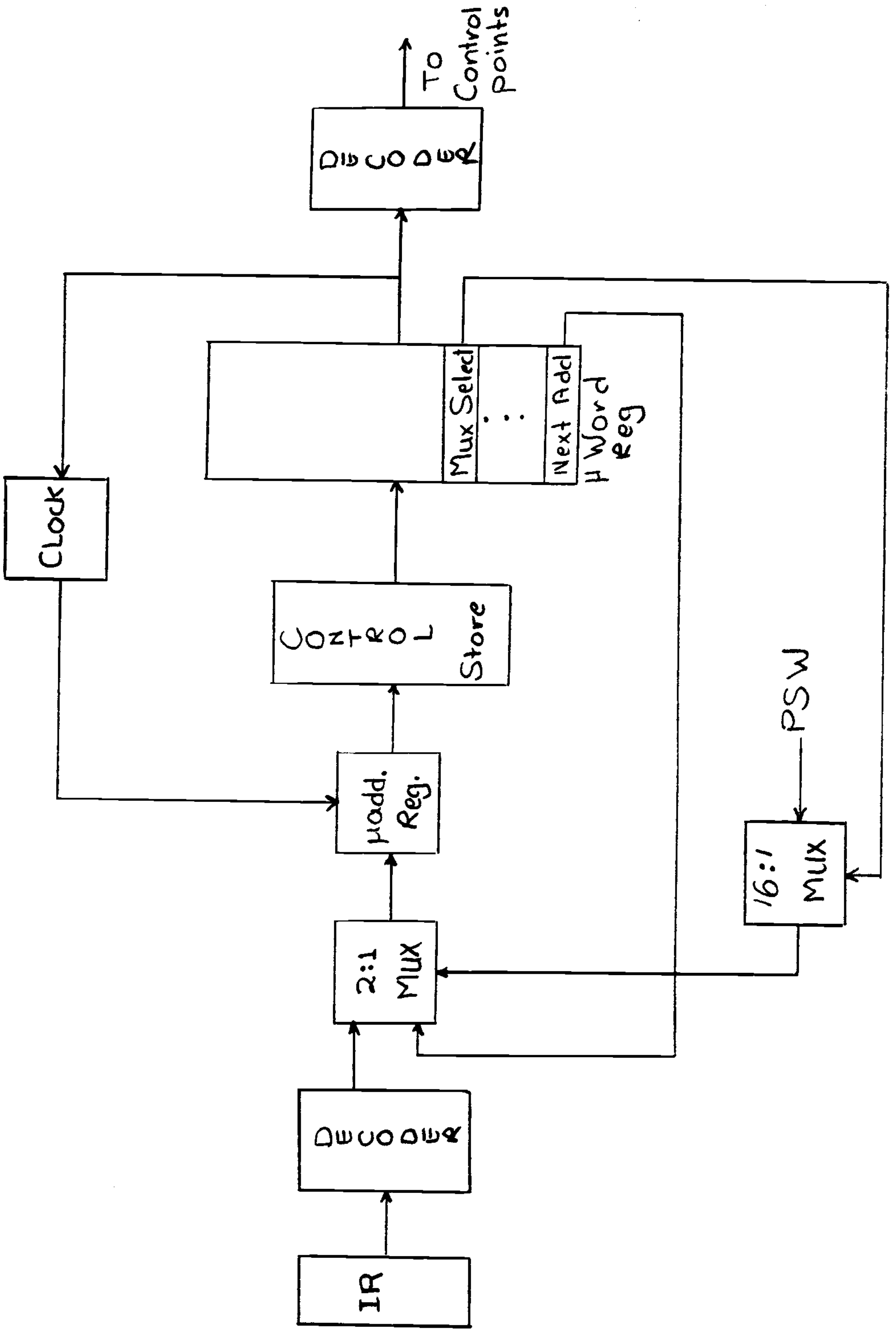
1.Five general purpose registers (GPR) with their input directly connected to the bus, while the output is connected via a tri-state buffer. GPR's are user addressable. The PC and SP are counters capable of counting up and down, the PC count down is not selected.

3.ALU, ACC, BUF, TEMP and PSW: one input of the ALU is directly tied to the bus, while the other is connected to the ACC, which in turn is tied to the bus. The output of the ALU is connected directly to BUF, which serves as a storage for the ALU result. BUF is tied to the bus via a tri-state buffer. Temp is a register used for storing intermediate values resulting from ALU operations. Temp is not a user addressable register. PSW is loaded with the different condition codes. Its output is directly connected to the condition code multiplexer of the controller.

4.Instruction register (IR): the IR is loaded from the bus, at the end of the fetch cycle, with the specific instruction to be executed. The output of IR is connected to the decoding PROM of the controller.

5.Main memory: when accessing main memory, two special registers are used; the memory address register (MAR) and the memory data register (MDR). The MAR is loaded from the bus, its ouput is connected directly to the memory. On the other hand, any data output from memory is loaded directly into MDR. MDR also accepts data from the bus, so to organize the operation, the input of MDR from the bus is fed into a tri-state buffer which has high impedence when MDR is currently accepting data from memory, otherwise it is enabled. A similar argument goes for MDR outputs, which are directly connected to the data input lines of memory, while tri-state buffered to the bus.

5.2.4 Controller: AF85 uses a microprogrammed control scheme, which provides flexibility in the implementation of the instruction set [19]. It also facilitates the addition of new instructions for future use. The control unit of AF85 is shown in Figure 3.



Control Unit of AF85

Figure 3

The contents of the micro-address register determine the current control unit state and are used to access the next micro-instruction word from the control store. Pulses from the control generator cause the loading of the micro-word and the micro-address registers with the next micro-word and micro-address respectively.

Most of the fields of the micro-word supply signals for conditioning and clocking the data paths. These fields require decoding circuits to identify the corresponding control signals.

The sequencing of microinstructions is implemented by adding a next address field in the micro-word, with the appropriate control. Now according to the multiplexer select, the next address of the microcode is selected from either the decoding PROM or from the control word.

5.2.5 Memory Organization: The main memory of AF85 is divided into three sections, start up routine, which is contained in a ROM, stack and user program and data storage.

Start up Routine: It is assumed that when we switch on, the PC and MAR are cleared instantaneously to start a fetch cycle of the first instruction in the start up routine. The steps of the start up routine are as follows:

```
        CLR     R0
        CLR     R1
```

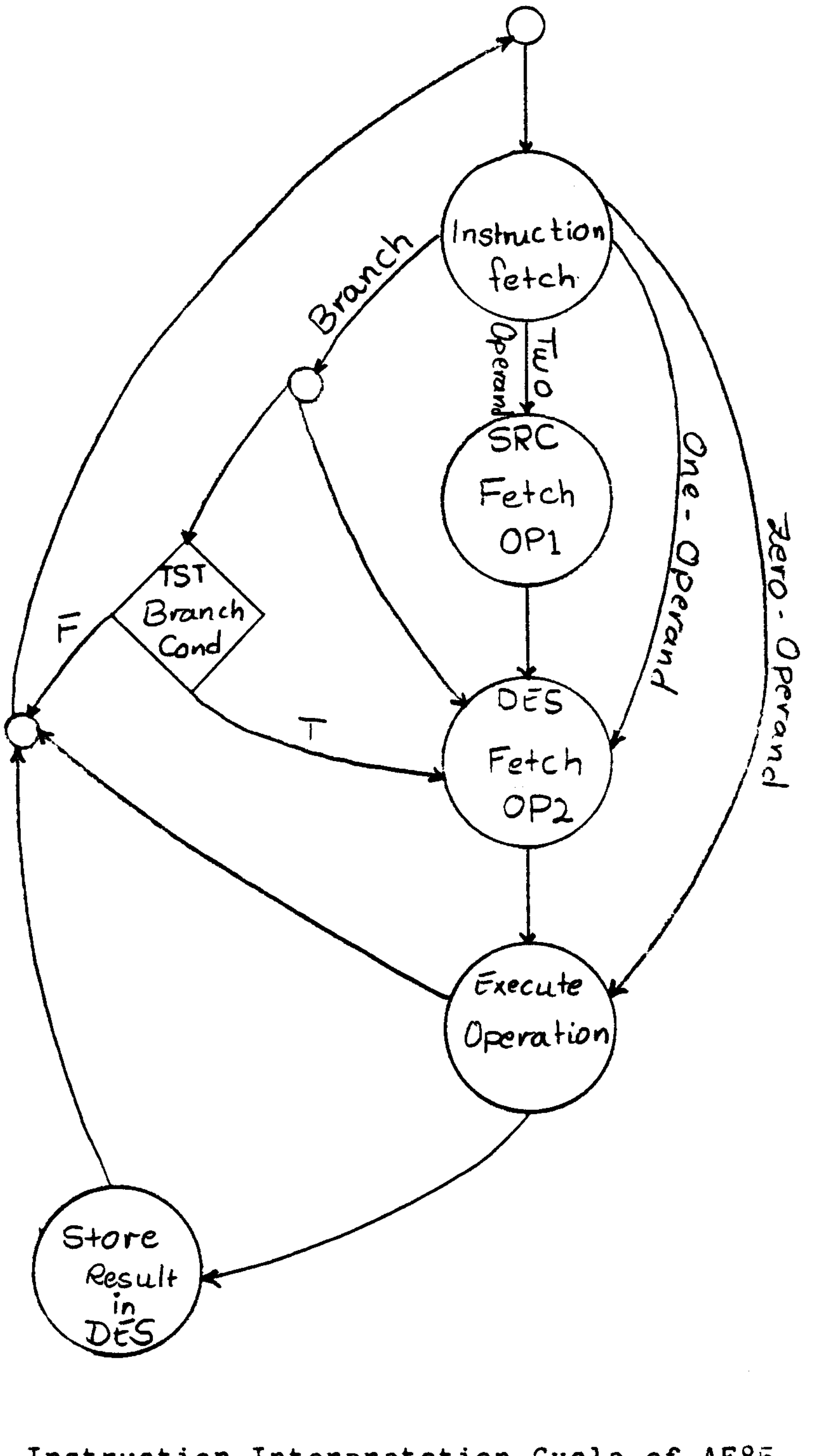
```
CLR      R2
CLR      R3
CLR      R4
CLR      TEMP
MOV      #1000,SP
MOV      #1000,PC
```

From above, CLR steps are done to assure a starting value of zero in all registers. The SP is loaded with a value which is equal to the top of the stack (TOS) plus one. At the end of the routine, the PC is loaded with the starting address of the user program and data storage.

The stack implemented in AF85 is a user stack which occupies part of the main memory. The user program and data storage is a defined space in memory which is user addressable.

5.3 Instruction Interpretation Cycle:

The instruction interpretation process of AF85 is shown in Figure 4. The process follows the common fetch-execute cycle.



Instruction Interpretation Cycle of AF85

Figure 4

VI. SIMULATION of AF85

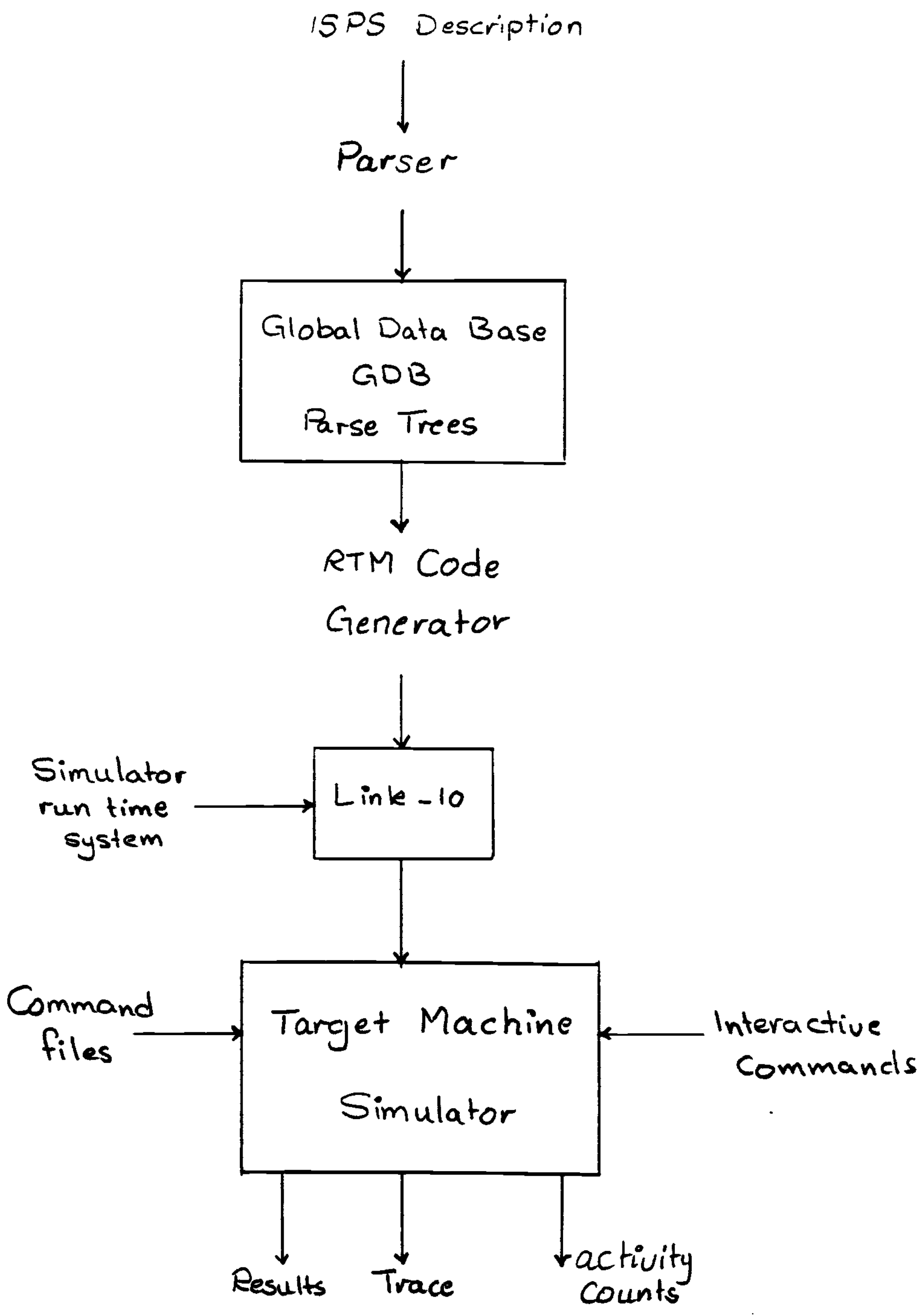
6.0 Introduction:

The design process begins by stating the problem, then demonstrating the solution by drawing a block diagram depicting the subsystems and the control/data paths. Once this stage is completed, the designer uses ISPS to describe the design. This description is then translated into a data base, which serves as a source for various other operations. Among these are system simulation at the description level, architecture evaluation and fault simulation

In this chapter, the ISPS simulator is used to refine the design of AF85 at the description level. Then it is used to evaluate the performance of the architecture via test programs. The performance measures derived are then compared with those of the PDP-8 and PDP-11 computer systems. Finally, the utility of the ISPS simulator in functional fault simulation is demonstrated by several examples.

6.1 The ISPS Simulator:

The process of using the ISPS simulator is shown in Figure 5,[7]. It starts by writing the ISPS description of AF85, a listing of the description is given in Appendix A. The ISPS parse tree is processed by a program GDBRTM. The GDBRTM program translates GDB files into code for an artificial machine called Register Transfer Machine (RTM). This code appears as a macro-10 file which must be linked with the simulator files. The reading of the GDB file, the generation of the RTM code file and using the macro-10 to assemble the RTM file are all done automatically.



Using ISPS Simulator

Figure 5

The user starts the process by compiling the ISPS description [6]. Then runs the GDBRTM program, and specifies the name of the GDB file. The output of this stage will have the same file name with extension REL. After the *.REL file has been generated, the RTM file is deleted automatically. Finally, the *.REL file is linked with the simulator files. The following example describes these steps.

```
.RUN ISPS
ISPS TRANSLATOR
*AF85

.

.RUN GDBRTM
GDB TO RTM TRANSLATOR
GDB FILE:AF85

.

.RUN LINK
*AF85
*@ISPSIM
*/SSAVE AF85
*/GO

.
```

Under normal operation, if no errors are detected, the above example describes the entire process. Errors that were not detected by the compiler will be detected by the GDBRTM translator. If these errors are serious, the whole operation will be aborted.

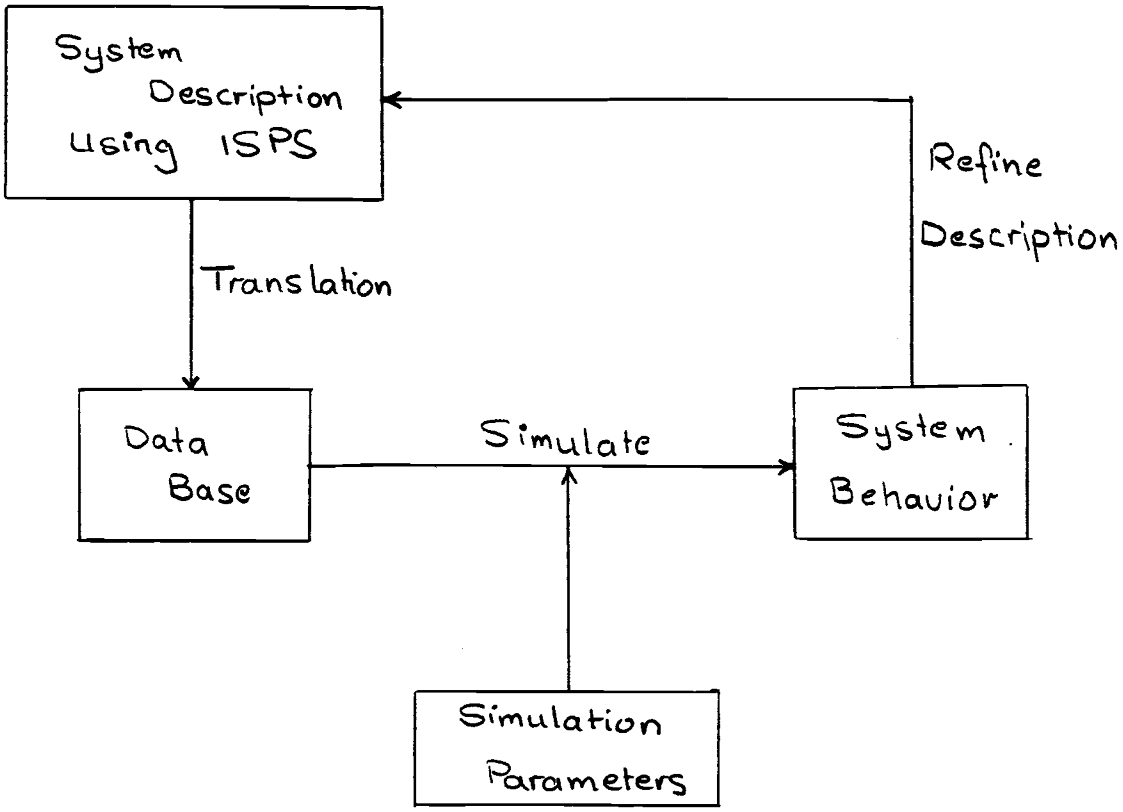
For more detailed information on the ISPS simulator, refer to the user's manual.

6.2 Simulation at the Description Level:

The advantage of simulating a digital system at the description level is two-fold, design verification and testing design alternatives [42]. It is much easier to change a line of code than it is to modify a breadboard circuit and its associated drawings and documentation.

6.2.1 Design Verification:

The original simulation of AF85 revealed an error in the branching structure. The branch instructions did not follow the defined FETCH-EXECUTE cycle properly. The error was corrected by rewriting the branch procedure of the system description, then simulating the system one more time. This process continues [42], Figure 6, until the desired system behavior is achieved.



Simulation at the Description Level

Figure 6

6.2.2 Design Alternatives:

The original design of AF85 used a decoding scheme of the instruction, to be executed, in which 10-bits of the Instruction Register (IR) were decoded at the same time by the same decoder [1]. This structure was simulated using the ISPS description outlined in Figure 7.

```
IR\INSTRUCTION.REGISTER<15:0>,
     OPCOD\OPERATION.CODE<5:0>::=IR<15:10>,
     S\SOURCE.FIELD<4:0>      ::=IR<9:5>,
          SRCMOD<1:0>         ::=S<4:3>,
          SRCREG<2:0>         ::=S<2:0>,
     D\DESTINATION.FIELD<4:0> ::=IR<4:0>,
          DESMOD<1:0>         ::=D<4:3>,
          DESREG<2:0>         ::=D<2:0>,
OPADM\OPCODE.ADDRESS.MOD<9:0> ::=IR<15:8> @ DESMOD<1:0>,
             .
             .
             .
EXECUTE():=
          BEGIN
          DECODE OPADM=>
                 BEGIN
                 0:=...
                 1:=...
                  .
                  .
                  .
                 END
          END,
```

The 10-bits Decoding Scheme I

Figure 7

A second alternative decoding scheme divides the 10-bits of the IR into two subfields, namely the OPCOD<5:0>:=IR<15:10> and addressing mode ADRMOD<3:0>:=SRCMOD<1:0> @ DESMOD<1:0>. Using this decoding scheme, the system was simulated using the ISPS description outlined in Figure 8.

```
IR\INSTRUCTION.REGISTER<15:0>,
     OPCOD\OPERATION.CODE<5:0>::=IR<15:10>,
     S\SOURCE.FIELD<4:0>      ::=IR<9:5>,
          SRCMOD<1:0>         ::=S<4:3>,
          SRCREG<2:0>         ::=S<2:0>,
     D\DESTINATION.FIELD<4:0> ::=IR<4:0>,
          DESMOD<1:0>         ::=D<4:3>,
          DESREG<2:0>         ::=D<2:0>,
ADRM\ADDRESSING.MODE<3:0>     ::=SRCMOD<1:0> @ DESMOD<1:0>,
             .
             .
             .
EXECUTE()::=
          BEGIN
          DECODE OPCOD=>
                 BEGIN
                 0\ADD::=ADD(),
                 1::=...
                  .
                  .
                  .
                 END
          END,
ADD()::=
      BEGIN
      GOPS\GET.OPERANDS(),
       .
       .
       .
      END,
 .
 .
 .
GOPS()::=
      BEGIN
      DECODE ADRM=>
             BEGIN
             0\IMMEDIATE::=BEGIN ... END,
             .
             .
             .
             END
      END,
```

The 10-bits Decoding Scheme II

Figure 8

A third alternative decoding scheme defines a new field, Type of Operation (TOP) in IR, TOP<1:0>::=IR<15:14>. It also separates the ADRMOD field into SRCMOD and DESMOD [44]. Thus to simulate this decoding scheme, the ISPS description outlined in Figure 9 is used.

```
IR\INSTRUCTION.REGISTER<15:0>,
     TOP\TYPE.OF.OPERATION<1:0>::=IR<15:14>,
     OPCOD\OPERATION.CODE<5:0> ::=IR<15:10>,
     S\SOURCE.FIELD<4:0>       ::=IR<9:5>,
          SRCMOD<1:0>          ::=S<4:3>,
          SRCREG<2:0>          ::=S<2:0>,
     D\DESTINATION.FIELD<4:0>  ::=IR<4:0>,
          DESMOD<1:0>          ::=D<4:3>,
          DESREG<2:0>          ::=D<2:0>,
              .
              .
              .
EXECUTE()::=
          BEGIN
          DECODE TOP=>
                  BEGIN
                  0\TWO.OPERAND ::=TWO(),
                  1\ZERO.OPERAND::=ZERO(),
                  2\ONE.OPERAND ::=ONE(),
                  3\BRANCH       ::=BRANCH(),
                 END
          END,
TWO()::=
      BEGIN
      DECODE OPCOD=>
             BEGIN
             0\ADD::=ADD(),
              .
              .
              .
             END
      END,
ZERO()::=BEGIN ... END,
ONE()::=BEGIN ... END,
BRANCH()::=BEGIN ... END,
  .
  .
  .
ADD()::=BEGIN
       OP1() NEXT
       OP2() NEXT
         .
         .
         .
       END,
  .
  .
  .
```

The 10-bits Decoding Scheme III

Figure 9

```
OP1():=
      BEGIN
      DECODE SRCMOD=>
                BEGIN
                O\IMMEDIATE:=BEGIN ... END,
                .
                .
                .
                END
      END,
  .
  .
  .
OP2():=
      BEGIN
      DECODE DESMOD=>
                BEGIN
                O\IMMEDIATE:=BEGIN ... END
                .
                .
                .
                END
      END,
  .
  .
  .
```

Figure 9 Continued

From above, it is clear that a number of design alternatives may be tested and verified quickly and easily using the ISPS simulator [6].

From the simulation runs, the third decoding scheme was chosen over the two other schemes. The structure proved to be faster, more efficient and easier to debug.

6.3 Performance Evaluation:

Traditionally, computer performance is evaluated by measuring the execution speed of a test program [44]. As a computer architecture does not specify the instruction execution times, the following alternative measures are defined [17]:

*S=number of bytes used to represent a test program.

*M=number of bytes transferred between primary memory and the processor during the execution of a test program.

*R=number of bytes transferred among internal registers of the processor during the execution of the test program.

The S measure indicates how well an architecture is suited for an application. A relatively small S measure is a good feature of an architecture.

The M and R measures characterize the bandwidth of the data paths between the processor and main memory and between the internal registers of the processor, respectively. A high M measure implies a large volume of information that has to be transferred, thus implying a slow instruction rate or a costly implementation. Similarly, a high R measure implies a slow instruction rate and a costy implemetation.

To evaluate the performance of AF85, two test programs were used [13],[17], [44]. The first program, character search, exercises the ability of a computer system to move through character strings sequentially. The second program, fibonacci number, tests the ability of the machine to support recursive routines.

Using the ISPS simulator, the two test programs were run on AF85, and the S, M, R measures were collected [10]. The simulator, while executing RTM codes, keeps count of all activities. These may be classified into three classes [6,7]:

*Counting the bits read from each register or memory location declared in the ISPS description.

*Counting the bits written into each register or memory location declared in the ISPS description.

*Counting the number of times each declared procedure or labeled statement in the ISPS description has been executed.

The S, M and R measures were collected using the appropriate counters [6].

To support the performance evaluation of AF85, the two test programs were run on the PDP-8 and PDP-11 descriptions provided by the ISPS software package. The S, M and R measures were collected. Table 4 lists the measures for the three different systems, for the two test programs.

Appendix B gives a listing of the two test programs running on AF85, PDP-8 and PDP-11.

SYSTEM	CHARACTER SEARCH			FIBONACCI NUMBER		
	S	M	R	S	M	R
AF85	106	826	3066	42	172	924
PDP-8	138	1602	3461	82	580	1278
PDP-11	78	566	2418	34	122	692

Table 4

Performance Measures

6.4 Functional Fault Simulation:

The ISPS simulator allows the specification of a variety of data and control faults at the functional level [6]. To completely specify a fault, it is necessary to indicate the specific type of fault to be inserted, where the fault is to occur in the simulation, when the fault is to take effect, and what other actions should be performed upon completion of each fault.

The fault insertion commands fall into three categories, which reflects the three major types of functional faults. These three groups are data faults, control faults, and operational faults. Each of these types of faults may be continuous or transient and may occur deterministically or probabilistically.

6.4.1 Data Faults:

Data faults are simulated by creating error conditions in registers, variables and memory locations. Two major types of data faults may be simulated, namely stuck-at fault and shorted fault.

To simulate a stuck-at-1 data fault in memory location 2010 of AF85, the following command is used,

```
DFAULT #100000 OR M[#2010]
```

To simulate the same fault with probability 25% and to print the value of PC each time the fault occurs, the following command is used,

```
DFAULT #100000 OR M[#2010]; PROB=25, DO VALUE PC$
```

To simulate a transient fault at memory location 2010, which takes effect after the fifth invokation of I.CYCLE and remains stuck for three invocations of the entity TOP, the following command is used,

```
DFAULT #100000 OR M[#2010]; WHEN I.CYCLE=5, FOR TOP=3
```

The continuous stuck-at-1 data fault in M[2010], may be detected by running the diagnostic program shown in Appendix C.

To simulate a shorted data fault between bit 0 of R1 and bit 15 of R2, the following command is used,

```
DFAULT R[#1]<0> OR R[#2]<15>
```

6.4.2 Control Faults:

Control faults may be simulated by creating error conditions in the ISPS 'DECODE' and 'IF' statements. For instance, to simulate the addressing mode of the source operand decoder of AF85, OP1, in such a way that it always selects the indirect addressing mode, the following command is used,

```
CFAULT 3, EQL; OP1(1)
OP1():=
     BEGIN
     DECODE SRCMOD=>
            BEGIN
            0:=...
            1:=...
            2:=...
            3:=...     !ALWAYS SELECTED
            END
     END
```

6.4.3 Operation Faults:

Operational faults may be simulated by injecting failures in arithmetic, logical and functional units of the system description. For instance, to simulate the failure of the ADD operation in AF85, the following command is used,

```
OFAULT ADD <= NOOP
```

Operational faults remain in effect until removed by the command DOFAULT. For more information on functional fault simulation, refer to the ISPS simulator manual.

VII. CONCLUSION

This dissertation demonstrated the utility of the ISPS HDL as a design tool. This goal was achieved by first presenting the current applications of ISPS in the area of automatic design of both hardware and software, then by using the notation to aid the design process of a new 16-bit processor, AF85.

In order to familiarize the reader with existing HDLs, chapter II presented an overview of a selected number of HDLs. The chapter concluded by defining a set of language features that were then used as a basis of comparison among the selected languages. ISPS was chosen because of the availability of a supporting software package that can be installed on the DEC-20 computer system. Moreover, ISPS had the biggest share of publications.

By evaluating the use of ISPS in the design process of AF85, the following conclusions were drawn:

1. ISPS proved to be useful in accurately describing the behavior of AF85 at the instruction level. Therefore, it is useful as a descriptive tool in teaching hardware design and computer architecture courses.
2. It is possible to verify the behavior of AF85 by writing the ISPS description of the system and then using it as an input to the simulator. The simulation indicated an error in the branching structure. This

error was corrected easily by rewriting the description of the BRANCH procedure. The simulation helped in revealing the error in a very early stage of design.

3. Three different decoding schemes of the IR were suggested in the design of AF85. ISPS simplified the task of examining all of them. By simulating the description of each alternative decoding scheme, it was possible to choose the more efficient one. Thus ISPS is a useful tool in studying design alternatives.
4. By using the ISPS descriptions of AF85, PDP-8 and PDP-11 systems, it was possible to conduct a performance evaluation study among the three systems. The performance of AF85 was acceptable. Hence, ISPS is useful in comparing and evaluating different architectures.
5. The ISPS simulator facilitates the insertion of faults into the simulation of AF85, thus observing the behavior of the faulted system. This type of study is helpful in evaluating prospective fault detection, diagnosis, recovery and repair mechanisms.

Two main disadvantages were noticed when using the ISPS software package. The first one is related to the implementation language. The ISPS compiler, translator and simulator are all written in BLISS, thus limiting the portability of the software. The second one is related to the documentation of the compiler's error messages. The result of the first compilation of the ISPS description of AF85 indicated many errors that were not clearly explained, hence these errors were corrected using trial and error techniques.

Overall, ISPS is a very useful design tool, supported by a high level simulator, which is interactive, capable, mature, widely used and accepted.

ISPS is the basis of several research projects on design automation. The latest uses ISPS descriptions as one of the inputs to a VLSI design automation system, which transforms architectural descriptions into layouts used for fabrication [23,24].

ISPS may very well serve as the basis of a register transfer level design automation system at Oregon State University (OSU). The language may also be used as a tool for teaching hardware design and architecture courses.

Finally, the information presented in this dissertation is meant to serve as a starting point to utilize the ISPS notation in the design environment of both hardware and software at OSU.

BIBLIOGRAPHY

(1) Asadi, A. and Sakallah, F.,"AF85 Minicomputer System", Term Project, Dept. of Electrical and Computer Engineering, Oregon State Univ., May 1985.

(2) Barbacci, M., Barnes, G., Cattell, R. and Siewiorek, D.,"The ISPS Computer Description Language," Dept. of Computer Science and Electrical Engineering, Carnegie-Mellon Univ., August 1979.

(3) Barbacci, M., Bell, C. and Newell, A.,"ISP: A Language to Describe Instruction Sets and Other Register Transfer Systems," COMPCON 72, pp. 227-230, 1972.

(4) Barbacci, M.,"A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Trans. Computers, Vol. C-24, No. 2, pp. 137-150, Feb. 1975.

(5) Barbacci, M.,"Instruction Set Processor Specifications for Simulation, Evaluation and Synthesis," Proc. 16th Annual Design Automation Conf., pp. 64-72, June 1979.

(6) Barbacci, M., Nagle, A. and Northcutt, J.," The Simbolic Manipulation of Computer Description: An ISPS Simulator," Dept. of Computer Science and Electrical Engineering, Carnegie-Mellon Univ., June 1981.

(7) Barbacci, M.,"Instruction Set Processor Specification (ISPS): The Notation and Its Applications," IEEE Trans. Computers, Vol. C-30, No. 1, pp. 24-40, Jan. 1981.

(8) Barbacci, M.,"Syntax and Semantics of CHDLs," Fifth Intl. Conf. on CHDLs and Their Applications, pp. 305-311, 1981.

(9) Barbacci, M.,"An Introduction to ISPS,"Dept. of Computer Science, Carnegie-Mellon Univ., Aug. 1982.

(10) Barbacci, M. et. al.,"An Architectural Research Facility- ISP Descriptions, Simulation, Data Collection," AFIPS Conf. Proc., Vol. 46, 1977, pp.161-173.

(11) Bell, C., Mudge, J. and McNamara, J.,Computer Engineering: A DEC View of Hardware Systems Design, Digital Press, 1978.

(12) Bell, C. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill Book Co., Inc., New York, N.Y., 1971.

(13) Benwell, N., Benchmarking: Computer Evaluation and Measurement, John Wiley and Sons, 1975.

(14) Breuer, M., "General Survey of Design Automation of Digital Computers," Proc. IEEE, Vol. 54, No. 12, pp. 1708-1721, Dec. 1966.

(15) Chu, Y., Computer Organization and Microprogramming, Prentice Hall, Inc., Englewood Cliffs, N.J., 1972.

(16) Chu, Y., "Introducing CDL," Computer, Vol. 7, No. 12, pp.31-33, Dec. 1974.

(17) Fuller, S. et. al., "Evaluation of Computer Architectures Via Test Programs," AFIPS Conf. Proc., Vol. 46, 1977, pp. 147-160.

(18) German, S. and Lieberherr, K., "ZUES: A Language for Expressing Algorithms in Hardware," Computer, Vol. 18, No. 12, Feb. 1985, pp. 55-65.

(19) Hamacher, V., Vranesic, Z. and Zaky, S., Computer Organization, McGraw-Hill Book Co., Inc., 1984.

(20) Hill, F. and Peterson, G., Digital Systems: Hardware Organization and Design, John Wiley and Sons, New York, N.Y., 1978.

(21) Hill, F., "Introducing AHPL," Computer, Vol. 7, No. 12, pp. 28-30, Dec. 1974.

(22) Hill, F. et. al., "Structural Specification with a Procedural HDL," IEEE Trans. Computers, Vol. c-30, No. 2, Feb. 1981, pp. 157-161.

(23) Kowalski, T., et. al., "The VLSI Design Automation Assistant: From Algorithms To Silicon," IEEE Design and Test, Aug. 1985, pp. 33-42.

(24) Kowalski, T. and Thomas, D., "The VLSI Design Automation Assistant: What's in a Knowledge Base," 22nd Annual Design Automation Conf., 1985, pp. 252-258.

(25) Lieberherr, K. and Knudsen, S., "ZUES: A Hardware Description Language for VLSI," 20th Design Automation Conf., Miami, Fla., June 1983, pp. 17-23.

(26) Lipovski, G., "Hardware Description Languages: Voices from the Tower of Babel," Computer, Vol. 10, No. 6,

pp. 14-17, June 1977.

(27) Lipsell, R., Marschner, E. and Shahdad, M.,"VHDL The Language," IEEE Design and Test, Vol. 3, No. 2, April 1986, pp. 28-41.

(28) Maissel, L. and Ofek, H.,"Hardware Design and Description Languages in IBM," IBM Journal of Research and Development, Vol. 28, no. 5, Sept. 1984, pp. 557-563.

(29) Maissel, L. and Ostapko, D.,"Interactive Design Language: A Unified Approach to Hardware Simulation, Synthesis and Documentation," 19th Design Automation Conf., Las Vegas, Nev., June 1982, pp. 193-201.

(30) Maissel, L. and Phoenix, R.,"IDL (Interactive Design Language) Features and Philosophy," IEEE Int'l Conf. Computer Design, Port Chester, N.Y., Sept. 1983, pp.667-669.

(31) Maryanski, F.,Digital Computer Simulation, Hayden Book Co., Inc., Rochelle Park, N.J., 1980.

(32) Milden, M.,"An Approach for Selecting a Language for Computer Hardware Description and Simulation," M.Sc. Thesis, Oregon State Univ., Feb. 1984.

(33) Nash, J.,"Bibliography of HDLs," ACM SIGDA Newsletter, Vol. 14, No. 1, Feb. 1984, pp. 18-37.

(34) Northcutt, J.,"The Design and Implementation of Fault Insertion Capabilities for ISPS," Proc. 17th Annual Design Automation Conf., pp. 197-209, 1980.

(35) Parker, A. et. al.,"The CMU Design System: An Example of Automated Data Path Design," Proc. 16th Annual Design Automation Conf., San Diego, CA., June 1979.

(36) Piloty, R., et. al.,"CONLAN A Formal Construction Method for Hardware Description Languages: Basic Principles," IFIPS National Computer Conf., pp. 209-217, 1980.

(37) Piloty, R. and Borrione, D.,"The CONLAN Project: Status and Future Plans," 19th Annual Design Automation Conf., Las Vegas, Nev., June 1982, pp. 202-212.

(38) Piloty, R. and Borrione, D.,"The CONLAN Project: Concepts, Implementation and Applications," IEEE Computer, Vol. 18, No. 2, Feb. 1985, pp. 81-90.

(39) Robinson, P. and Dion J.," Programming Languages for Hardware Description," Proc. 20th Annual Design Automation Conf., pp. 12-16, 1983.

(40) Sakallah, F.,"Computer Hardware Description Languages: A Survey," Term Paper, Dept. of Electrical and Computer Engineering, Oregon State Univ., Feb. 1984.

(41) Shahdad, M., et. al.,"VHSIC Hardware Description Language," IEEE Computer, Vol. 18, No. 2, Feb. 1985, pp. 94-103.

(42) Shiva, S.,"Computer Hardware Description Languages- A Tutorial," Proc. of the IEEE, Vol. 67, No. 12, pp. 1605-1615, Dec. 1979.

(43) Siewiorek, D.,"Introducing ISP," Computer, Vol. 7, No. 12, Dec. 1974, pp. 39-41.

(44) Siewiorek, D., Bell, C. and Newell, A.,<u>Computer Structures: Principles and Examples</u>, McGraw-Hill Book Co., Inc., New York, 1982.

(45) Siewiorek, D. and Barbacci, M.,"The CMU RT-CAD System- An Innovative Approach to Computer Aided Design," Proc. of AFIPS National Computer Conf., Vol. 45, 1976.

(46) Singh, A. and Tracey, J.,"Development of Comparison Features for Computer Hardware Description Languages," Fifth Int'l Conf. CHDLs and Their Applications, pp. 247-263, 1981.

(47) VanCleemput, W.,"Computer Hardware Description Languages and Their Applications," Proc. 16th Annual Design Automation Conf., pp. 554-560, 1979.

APPENDICES

APPENDIX A

ISPS Description of AF85

```
AF85:=
BEGIN
** Memory.State **
M\Memory[0:32767]<15:0>,
!
** Processor.State **
ACC\ACCUMULATOR<15:0>
TEMP<15:0>,
MDR<15:0>,
MAR<15:0>,
BUF\BUFFER<15:0>,
PSW\PROCESSOR.STATUS.WORD<3,0>,
    C\CARRY<>:=PSW<0>,
    V\OVERFLOW<>:=PSW<1>,
    Z\ZERO<>:=PSW<2>,
    N\NEGATIVE<>:=PSW<3>,
R[6:0]<15:0>,
    R0<15:0>:=R[0]<15:0>,
    R1<15:0>:=R[1]<15:0>,
    R2<15:0>:=R[2]<15:0>,
    R3<15:0>:=R[3]<15:0>,
    R4<15:0>:=R[4]<15:0>,
    PC<15:0>:=R[5]<15:0>,
    SP<15:0>:=R[6]<15:0>,
IR\INSTRUCTION.REGISTER<15:0>,
    TOP\TYPE.OF.OPERATION<1:0>::=IR<15:14>,
    OPCOD\OPERATION.CODE<5:0> ::=IR<15:10>,
    S\SOURCE.FIELD<4:0>       ::=IR<9:5>,
      SRCMOD\SOURCE.MODE<1:0> ::=S<4:3>,
      SRCREG\SOURCE.REG<2:0>  ::=S<2:0>,
    D\DESTINATION.FIELD<4:0>  ::=IR<4:0>,
      DESMOD\DESTIN.MODE<1:0> ::=D<4:3>,
      DESREG\DESTIN.REG<2:0>  ::=D<2:0>,
!
** Fetch.Cycle **
I.CYCLE:=
       BEGIN
       REPEAT BEGIN
              IR=M[PC] NEXT
              PC=PC+1  NEXT
              EXECUTE()
              END
       END,
!
** Execute.Cycle **
EXECUTE():=
       BEGIN
       DECODE TOP=>
              BEGIN
              0\TWO.OPERAND :=TWO(),
              1\ZERO.OPERAND:=ZERO(),
              2\ONE.OPERAND :=ONE(),
              3\BRANCH      :=BRANCH(),
               END
        END,
!
** Two.Operand.Instructions **
TWO():=
        BEGIN
        DECODE OPCODE=>
               BEGIN
               0\ADD    :=ADD(),
               1\SUB    :=SUB(),
               2\CMP    :=CMP(),
               3\CAND   :=CAND(),
               4\COR    :=COR(),
               5\CXOR   :=CXOR(),
               6\MOV    :=MOV(),
               7\SHR    :=SHR(),
               8\SHL    :=SHL(),
               OTHERWISE:=NO.OP
               END
        END,
!
** Zero.Operand.Instructions **
ZERO():=
        BEGIN
        DECODE OPCODE=>
               BEGIN
               16\NO.OPERATION:=NO.OP,
               17\RET         :=RET(),
               18\STACK.ADD:=STKADD(),
               19\STACK.SUB:=STKSUB(),
               20\STACK.OR :=STKOR(),
               21\STACK.AND:=STKAND(),
               22\STACK.XOR:=STKXOR(),
               23\STACK.CMP:=STKCMP(),
               24\HLT      :=STOP(),
               OTHERWISE   :=NO.OP
               END
        END,
!
** One.Operand.Instructions **
ONE():=
        BEGIN
        DECODE OPCODE=>
               BEGIN
               32\CLEAR :=CLR(),
               33\PUSH  :=PUSH(),
               34\POP   :=POP(),
               35\CALL  :=CALL(),
               OTHERWISE:=NO.OP
               END
        END,
!
** Branch.Instructions **
```

```
BRANCH():=
       BEGIN
       DECODE OPCOD=>
              BEGIN
              48        ::=BR(),
              49        ::=BEQ(),
              50        ::=BMI(),
              51        ::=BCS(),
              52        ::=BVS(),
              53        ::=BLT(),
              54        ::=BLE(),
              55        ::=BLO(),
              56        ::=BNE(),
              57        ::=BP(),
              58        ::=BNC(),
              59        ::=BNV(),
              OTHERWISE::=NO.OP
              END
       END,
!
!Fetching Source Operand
OP1():=
       BEGIN
       DECODE SRCMOD=>
              BEGIN
              0\IMMEDIATE::=BEGIN
                             MDR=M[PC] NEXT
                             MAR=PC ; PC=PC+1
                             END,
              1\REGISTER ::=ACC=R[SRCREG],
              2\MEMORY    ::=BEGIN
                             MDR=M[PC] NEXT
                             PC=PC+1 ; ACC=R[SRCREG] NEXT
                             BUF=ACC+MDR ; MAR=BUF NEXT
                             MDR=M[MAR]
                             END,
              3\INDIRECT ::=BEGIN
                             MDR=M[PC] NEXT
                             PC=PC+1 ; MAR=MDR NEXT
                             MDR=M[MAR]
                             END,
              END
       END,
!
!Fetching Destination Operand
OP2():=
       BEGIN
       DECODE DESMOD=>
         BEGIN
         0\IMMEDIATE::=STOP(),
         1\REGISTER ::=BEGIN
              DECODE SRCMOD=>
                 BEGIN
```

```
                1\REGISTER::=MDR=R[DESREG],
                OTHERWISE ::=ACC=R[DESREG]
                END
        END,
        2\MEMORY   ::=BEGIN
             DECODE SRCMOD=>
                BEGIN
                1\REGISTER::=TEMP=ACC,
                OTHERWISE ::=BEGIN
                          TEMP=MDR ; MDR=M[PC] NEXT
                          PC=PC+1 NEXT
                          ACC=R[DESREG] NEXT
                          BUF=ACC+MDR ; MAR=BUF NEXT
                          MDR=M[MAR] ; ACC=TEMP
                            END
                END
                      END,
        3\INDIRECT ::=BEGIN
             DECODE SRCMOD=>
                BEGIN
                1\REGISTER::=TEMP=ACC,
                OTHERWISE ::=BEGIN
                            TEMP=MDR ; MDR=M[PC] NEXT
                            PC=PC+1 ; MAR=MDR NEXT
                            MDR=M[MAR] ; ACC=TEMP
                                             END
                                 END
                          END
             END
      END,
!
!Executing Two Operand Instructions
ADD():=
    BEGIN
    OP1() NEXT
    OP2() NEXT
    C@BUF=ACC+MDR NEXT
    V=(MDR<15> EQV ACC<15>) AND (MDR<15> XOR BUF<15>) NEXT
    Z=BUF EQL 0 ; N=BUF<15> NEXT
    DECODE DESMOD=>
           BEGIN
           1\REGISTER::=R[DESREG]=BUF,
           OTHERWISE ::=M[MAR]=BUF
           END
    END,
!
SUB():=
    BEGIN
    OP1() NEXT
    OP2() NEXT
    C@BUF=ACC-MDR NEXT
    V=(MDR<15> XOR ACC<15>) AND (MDR<15> EQV BUF<15>) NEXT
    Z=BUF EQL 0 ; N=BUF<15> NEXT
    DECODE DESMOD=>
           BEGIN
           1\REGISTER:=R[DESREG]=BUF,
           OTHERWISE :=M[MAR]=BUF
           END
    END,
!
CAND():=
       BEGIN
       OP1() NEXT
       OP2() NEXT
       C@BUF=ACC AND MDR NEXT
       V=0 ; Z=BUF EQL 0 ; N=BUF<15> NEXT
       DECODE DESMOD=>
              BEGIN
              1\REGISTER:=R[DESREG]=BUF,
              OTHERWISE :=M[MAR]=BUF
              END
       END,
!
COR():=
       BEGIN
       OP1() NEXT
       OP2() NEXT
       C@BUF=ACC OR MDR NEXT
       V=0 ; Z=BUF EQL 0 ; N=BUF<15> NEXT
       DECODE DESMOD=>
              BEGIN
              1\REGISTER:=R[DESREG]=BUF,
              OTHERWISE :=M[MAR]=BUF
              END
       END,
!
CXOR():=
       BEGIN
       OP1() NEXT
       OP2() NEXT
       C@BUF=ACC XOR MDR NEXT
       V=0 ; Z=BUF EQL 0 ; N=BUF<15> NEXT
       DECODE DESMOD=>
              BEGIN
              1\REGISTER:=R[DESREG]=BUF,
              OTHERWISE :=M[MAR]=BUF
              END
       END,
!
MOV():=
       BEGIN
       OP1() NEXT
       OP2() NEXT
       DECODE DESMOD=>
              BEGIN
              1\REGISTER:=R[DESREG]=MDR,
                OTHERWISE :=M[MAR]=MDR
                END
           END,
!
CMP():=
        BEGIN
        OP1() NEXT
        OP2() NEXT
        C@BUF=ACC-MDR NEXT
        V=(BUF<15> EQV MDR<15>) AND (ACC<15> XOR MDR<15>) NEXT
        Z=BUF EQL 0 ; N=BUF<15>
        END,
!
SHL():=
           BEGIN
           OP1() NEXT
           OP2() NEXT
           C@BUF=MDR SLO ACC NEXT
           V=0 ; Z=BUF EQL 0 ; N=BUF<15> NEXT
           DECODE DESMOD=>
                BEGIN
                1\REGISTER:=R[DESREG]=BUF,
                OTHERWISE :=M[MAR]=BUF
                END
           END,
!
SHR():=
           BEGIN
           OP1() NEXT
           OP2() NEXT
           C@BUF=MDR SRO ACC NEXT
           V=0 ; Z=BUF EQL 0 ; N=BUF<15> NEXT
           DECODE DESMOD=>
                BEGIN
                1\REGISTER:=R[DESREG]=BUF,
                OTHERWISE :=M[MAR]=BUF
                END
           END,
!
!Executing Zero Operand Instructions
RET():=
           BEGIN
           MDR=M[SP] NEXT
           SP=SP+1 ; PC=MDR
           END,
!
STKADD():=
        BEGIN
        MDR=M[SP] ; SP=SP+1 ;
        ACC=MDR ; MDR=M[SP] ;
        C@BUF=ACC+MDR ;
        V=(MDR<15> EQV ACC<15>) AND (MDR<15> XOR BUF<15>) ;
        Z=BUF EQL 0 ; N=BUF<15> ;
```

```
    M[SP]=BUF
    END,
!
STKSUB():=
    BEGIN
    MDR=M[SP] ; SP=SP+1 ;
    ACC=MDR ; MDR=M[SP] ;
    C@BUF=ACC-MDR ;
    V=(MDR<15> XOR ACC<15>) AND (MDR<15> EQV BUF<15>) ;
    Z=BUF EQL 0 ; N=BUF<15> ;
    M[SP]=BUF
    END,
!
STKOR():=
       BEGIN
       MDR=M[SP] ; SP=SP+1 ;
       ACC=MDR ; MDR=M[SP] ;
       C@BUF=ACC OR MDR ;
       V=0 ; Z=BUF EQL 0 ; N=BUF<15> ;
       M[SP]=BUF
       END,
!
STKAND():=
       BEGIN
       MDR=M[SP] ; SP=SP+1 ;
       ACC=MDR ; MDR=M[SP] ;
       C@BUF=ACC AND MDR ;
       V=0 ; Z=BUF EQL 0 ; N=BUF<15> ;
       M[SP]=BUF
       END,
!
STKXOR():=
       BEGIN
       MDR=M[SP] ; SP=SP+1 ;
       ACC=MDR ; MDR=M[SP] ;
       C@BUF=ACC XOR MDR ;
       V=0 ; Z=BUF EQL 0 ; N=BUF<15> ;
       M[SP]=BUF
       END,
!
STKCMP():=
    BEGIN
    MDR=M[SP] ; SP=SP+1 ;
    ACC=MDR ; MDR=M[SP] ;
    C@BUF=MDR-ACC ;
    V=(BUF<15> EQV MDR<15>) AND (ACC<15> XOR MDR<15>) ;
    Z=BUF EQL 0 ; N=BUF<15> ;
    SP=SP-1
    END,
!
!Executing One Operand Instructions
CLR():=
       BEGIN
```

```
        OP1() NEXT
        MDR=0 ; ACC=0 ;
        DECODE SRCMOD=>
                BEGIN
                1\REGISTER:=R[SRCREG]=ACC,
                OTHERWISE :=M[MAR]=MDR
                END
        END,
!
PUSH():=
        BEGIN
        OP1() NEXT
        SP=SP-1 ;
        DECODE SRCMOD=>
                BEGIN
                1\REGISTER:=M[SP]=ACC,
                OTHERWISE :=M[SP]=MDR
                END
        END,
!
POP():=
        BEGIN
        OP1() NEXT
        TEMP=M[SP] ; SP=SP+1 ;
        DECODE SRCMOD=>
                BEGIN
                1\REGISTER:=RSRCREG]=TEMP,
                OTHERWISE :=M[MAR]=TEMP
                END
        END,
!
CALL():=
        BEGIN
        OP1() NEXT
        SP=SP-1 ; M[SP]=PC ;
        PC=MDR
        END,
!
!Executing Branch Instructions
BR():=
        BEGIN
        OP1() NEXT
        PC=MAR
        END,
!
BEQ():=
        BEGIN
        DECODE Z=>
                0:=PC=PC+1,
                1:=OP1() NEXT
                   PC=MAR,
        END,
!
```

```
BMI():=
        BEGIN
        DECODE N=>
                0:=PC=PC+1,
                1:=OP1() NEXT
                   PC=MAR,
        END,
!
BCS():=
        BEGIN
        DECODE C=>
                0:=PC=PC+1,
                1:=OP1() NEXT
                     PC=MAR
                     END
        END,
!
BVS():=
        BEGIN
        DECODE V=>
                0:=PC=PC+1,
                1:=OP1() NEXT
                   PC=MAR,
        END,
!
BLT():=
        BEGIN
        DECODE (N XOR V)=>
                         0:=PC=PC+1,
                         1:=OP1() NEXT
                            PC=MAR,
        END,
!
BLE():=
        BEGIN
        DECODE (Z OR (N XOR V))=>
                                  0:=PC=PC+1,
                                  1:=OP1() NEXT
                                     PC=MAR,
        END,
!
BLO():=
        BEGIN
        DECODE (C OR Z)=>
                         0:=PC=PC+1,
                         1:=OP1() NEXT
                            PC=MAR,
        END,
!
BNE():=
        BEGIN
        DECODE Z=>
                 0:= OP1() NEXT
```

```
                    PC=MAR,
                1:=PC=PC+1
       END,
!
BP():=
       BEGIN
       DECODE N=>
                0:=OP1() NEXT
                   PC=MAR,
                1:=PC=PC+1,
       END,
!
BNC():=
       BEGIN
       DECODE C=>
                0:=OP1() NEXT
                   PC=MAR,
                1:=PC=PC+1,
       END,
!
BNV():=
       BEGIN
       DECODE V=>
                0:=OP1() NEXT
                   PC=MAR,
                1:=PC=PC+1,
       END,
END
```

APPENDIX B

Listings of Simulation Command Files

of

AF85, PDP-8, PDP-11

```
!SIMULATION COMMAND FILE FOR AF85
!FIBONACCI NUMBER
ECHO
BTRACE AF85
PC=#2000
S M[#2000]=#100440        !CLR R1
S M[#2001]=#100400        !CLR R0
S M[#2002]=#11            !ADD #1,R1
S M[#2003]=#1
S M[#2004]=#10            !ADD #1,R0
S M[#2005]=#1
S M[#2006]=#15413         !MOV @K,R3
S M[#2007]=#3000          !K
S M[#2010]=#14015         !MOV #3,R5
S M[#2011]=#3
S M[#2012]=#14412         !LOOP:MOV R0,R2
S M[#2013]=#450           !ADD R1,R0
S M[#2014]=#14511         !MOV R2,R1
S M[#2015]=#15            !ADD #1,R5
S M[#2016]=#1
S M[#2017]=#4555          !CMP R3,R5
S M[#2020]=#163200        !BP  LOOP
S M[#2021]=#177760
S M[#2022]=#14430         !MOV R0,@FIB
S M[#2023]=#3001          !FIB
S M[#2024]=#60000         !STOP
S M[#3000]=#31            !K
START I.CYCLE
V R[#0:#5]
EXIT
```

```
!SIMULATION COMMAND FILE FOR AF85
!CHARACTER SEARCH
ECHO
BTRACE AF85
PC=#5000
S M[#5000]=#15410         !MOV @N,R0
S M[#5001]=#266           !N
S M[#5002]=#15411         !MOV @M,R1
S M[#5003]=#267           !M
S M[#5004]=#2411          !SUB R0,R1
S M[#5005]=#100400        !CLR R0
S M[#5006]=#100500        !CLR R2
S M[#5007]=#14413         !LOOP:MOV R0,R3
S M[#5010]=#4450          !CMP R1,R0
S M[#5011]=#145200        !BMI T1
S M[#5012]=#1
S M[#5013]=#141200        !BR CONT
S M[#5014]=#10            !OFFSET
S M[#5015]=#4012          !T1:CMP #0,R2
S M[#5016]=#0
S M[#5017]=#143200        !BEQ T2
S M[#5020]=#3
S M[#5021]=#141200        !BR CONT
S M[#5022]=#2
S M[#5023]=#141200        !T2:BR END
S M[#5024]=#27
S M[#5025]=#5022          !CONT:CMP STRING(R0),SUBS(R2)
S M[#5026]=#2000          !STRING
S M[#5027]=#4000          !SUBS
S M[#5030]=#161200        !BNE L1
S M[#5031]=#12
S M[#5032]=#10            !ADD #1,R0
S M[#5033]=#1
S M[#5034]=#12            !ADD #1,R2
S M[#5035]=#1
S M[#5036]=#5412          !CMP @M,R2
S M[#5037]=#267           !M
S M[#5040]=#143200        !BEQ L2
S M[#5041]=#14
S M[#5042]=#141200        !BR L3
S M[#5043]=#4
S M[#5044]=#13            !L1:ADD #1,R3
S M[#5045]=#1
S M[#5046]=#14550         !MOV R3,R0
S M[#5047]=#100500        !CLR R2
S M[#5050]=#5410          !L3:CMP @N,R0
S M[#5051]=#266           !N
S M[#5052]=#163200        !BP LOOP
S M[#5053]=#177733
S M[#5054]=#100640        !END:CLR R5
S M[#5055]=#60000         !STOP
S M[#5056]=#2430          !L2:SUB R0,@M
S M[#5057]=#267           !M
S M[#5060]=#15415         !MOV @M,R5
```

```
S M[#5061]=#267           !M
S M[#5062]=#15            !ADD #STRING,R5
S M[#5063]=#2000          !STRING
S M[#5064]=#60000         !STOP
S M[#266]=#24             !N
S M[#267]=#12             !M
S M[#2000]=#101           !STRING
S M[#2001]=#102
S M[#2002]=#103
S M[#2003]=#104
S M[#2004]=#105
S M[#2005]=#106
S M[#2006]=#107
S M[#2007]=#110
S M[#2010]=#111
S M[#2011]=#112
S M[#2012]=#113
S M[#2013]=#114
S M[#2014]=#115
S M[#2015]=#116
S M[#2016]=#117
S M[#2017]=#120
S M[#2020]=#121
S M[#2021]=#122
S M[#2022]=#123
S M[#2023]=#124
S M[#4000]=#105           !SUBS
S M[#4001]=#106
S M[#4002]=#107
S M[#4003]=#110
S M[#4004]=#111
S M[#4005]=#112
S M[#4006]=#113
S M[#4007]=#114
S M[#4010]=#115
S M[#4011]=#116
START I.CYCLE
V R[#0:R5]
EXIT
```

```
!SIMULATION COMMAND FILE FOR PDP8
!FIBONACCI NUMBER
ECHO
RADIX OCTAL
BTRACE PDP8
S PC=#200
S M[#200]=#7600             !CLA
S M[#201]=#3301             !DCA R1
S M[#202]=#3300             !DCA R0
S M[#203]=#7240             !STA
S M[#204]=#0310             !AND ONE
S M[#205]=#1301             !TAD R1
S M[#206]=#3301             !DCA R1
S M[#207]=#7240             !STA
S M[#210]=#0310             !AND ONE
S M[#211]=#1300             !TAD R0
S M[#212]=#3300             !DCA R0
S M[#213]=#7240             !STA
S M[#214]=#0306             !AND K
S M[#215]=#3303             !DCA R3
S M[#216]=#7240             !STA
S M[#217]=#0305             !AND THREE
S M[#220]=#3304             !DCA R0
S M[#221]=#7240             !LOOP:STA
S M[#222]=#0300             !AND R0
S M[#223]=#3302             !DCA R2
S M[#224]=#7240             !STA
S M[#225]=#0301             !AND R1
S M[#226]=#1300             !TAD R0
S M[#227]=#3300             !DCA R0
S M[#230]=#7240             !STA
S M[#231]=#0302             !AND R2
S M[#232]=#3301             !DCA R1
S M[#233]=#7240             !STA
S M[#234]=#0310             !AND ONE
S M[#235]=#1304             !TAD R4
S M[#236]=#3304             !DCA R4
S M[#237]=#7240             !STA
S M[#240]=#0304             !AND R4
S M[#241]=#7041             !CIA
S M[#242]=#1303             !TAD R3
S M[#243]=#7500             !SMA
S M[#244]=#5221             !JMP LOOP
S M[#245]=#7240             !STA
S M[#246]=#0300             !AND R0
S M[#247]=#3307             !DCA FIB
S M[#250]=#7402             !HLT
!M[#300]=R0
!M[#301]=R1
!M[#302]=R2
!M[#303]=R3
!M[#304]=R4
S M[#305]=#3                !THREE
```

```
S M[#306]=#6            !K
!M[#307]=FIB
S M[#310]=#1            !ONE
START INTERPRET
V M[#300:#310]
EXIT
```

```
!SIMULATION COMMAND FILE FOR PDP8
!CHARACTER SEARCH
ECHO
RADIX OCTAL
BTRACE PDP8
S PC=#200
S M[#200]=#7240 !STA
S M[#201]=#0311 !AND M
S M[#202]=#7041 !CIA
S M[#203]=#1310 !TAD N
S M[#204]=#3312 !DCA N-M
S M[#205]=#7240 !LOOP:STA
S M[#206]=#0316 !AND CTNM
S M[#207]=#3317 !DCA CTK
S M[#210]=#7240 !STA
S M[#211]=#0316 !ADN CTNM
S M[#212]=#7041 !CIA
S M[#213]=#1312 !TAD N-M
S M[#214]=#7510 !SPA
S M[#215]=#5217 !JMP T1
S M[#216]=#5227 !JMP CONT
S M[#217]=#7240 !T1:STA
S M[#220]=#0315 !AND CTM
S M[#221]=#7041 !CIA
S M[#222]=#1323 !TAD ZERO
S M[#223]=#7450 !SNA
S M[#224]=#5226 !JMP T2
S M[#225]=#5227 !JMP CONT
S M[#226]=#5274 !T2:JMP END
S M[#227]=#7240 !CONT:STA
S M[#230]=#0713 !AND @R1
S M[#231]=#7041 !CIA
S M[#232]=#1714 !TAD @R2
S M[#233]=#7440 !SZA
S M[#234]=#5250 !JMP L1
S M[#235]=#2313 !ISZ R1
S M[#236]=#2314 !ISZ R2
S M[#237]=#2315 !ISZ CTM
S M[#240]=#2316 !ISZ CTNM
S M[#241]=#7240 !STA
S M[#242]=#0311 !AND M
S M[#243]=#7041 !CIA
S M[#244]=#1315 !TAD CTM
S M[#245]=#7500 !
S M[#246]=#5277 !JMP L2
S M[#247]=#5266 !JMP L3
S M[#250]=#2317 !L1:ISZ CTK
S M[#251]=#7240 !STA
S M[#252]=#0317 !AND CTK
S M[#253]=#3316 !DCA CTNM
S M[#254]=#7240 !STA
S M[#255]=#0317 !AND CTK
S M[#256]=#1320 !TAD ADSTRING
S M[#257]=#3313 !DCA R1
```

```
S M[#260]=#7240 !STA
S M[#261]=#0321 !AND ADSUBS
S M[#262]=#3314 !DCA R2
S M[#263]=#7240 !STA
S M[#264]=#0322 !AND ONE
S M[#265]=#3315 !DCA CTM
S M[#266]=#7240 !L3:STA
S M[#267]=#0316 !AND CTNM
S M[#270]=#7041 !CIA
S M[#271]=#1310 !TAD N
S M[#272]=#7500 !SMA
S M[#273]=#5205 !JMP LOOP
S M[#274]=#7600 !END:CLA
S M[#275]=#3362 !DCA RESULT
S M[#276]=#7402 !HLT
S M[#277]=#7240 !L2:STA
S M[#300]=#0311 !AND M
S M[#301]=#7041 !CIA
S M[#302]=#1313 !TAD R1
S M[#303]=#3362 !DCA RESULT
S M[#304]=#7402 !HLT
!DATA
S M[#310]=#24   !N
S M[#311]=#12   !M
!M[#312]=N-M
S M[#313]=#324  !R1
S M[#314]=#350  !R2
S M[#315]=#0    !J
S M[#316]=#0    !I
S M[#317]=#1    !K
S M[#320]=#324  !ADSTRING
S M[#321]=#350  !ADSUBS
S M[#322]=#0    !ONE
S M[#323]=#0    !ZERO
S M[#324]=#101  !STRING
S M[#325]=#102
S M[#326]=#103
S M[#327]=#104
S M[#330]=#105
S M[#331]=#106
S M[#332]=#107
S M[#333]=#110
S M[#334]=#111
S M[#335]=#112
S M[#336]=#113
S M[#337]=#114
S M[#340]=#115
S M[#341]=#116
S M[#342]=#117
S M[#343]=#120
S M[#344]=#121
S M[#345]=#122
S M[#346]=#123
S M[#347]=#124
S M[#350]=#105  !SUBS
```

```
S M[#351]=#106
S M[#352]=#107
S M[#353]=#110
S M[#354]=#111
S M[#355]=#112
S M[#356]=#113
S M[#357]=#114
S M[#360]=#115
S M[#361]=#116
!M[#362]=RESULT
START INTERPRET
V M[#362]
EXIT
```

```
!SIMULATION COMMAND FILE FOR PDP11
!FIBANACCI NUMBER
ECHO
RADIX OCTAL
S R[#7]=#1000
S MW[#1000]=#5001          !START CLR R1
S MW[#1002]=#5000          !CLR R0
S MW[#1004]=#5201          !INC R1
S MW[#1006]=#5200          !INC R0
S MW[#1010]=#16703         !MOV K,R3
S MW[#1012]=#26            !K
S MW[#1014]=#12704         !MOV #3,R4
S MW[#1016]=#3
S MW[#1020]=#10002         !LOOP:MOV R0,R2
S MW[#1022]=#60100         !ADD R1,R0
S MW[#1024]=#10201         !MOV R2,R1
S MW[#1026]=#5204          !INC R4
S MW[#1030]=#20304         !CMP R3,R4
S MW[#1032]=#2372          !BGE LOOP
S MW[#1034]=#10067         !MOV R0,FIB
S MW[#1036]=#4
S MW[#1040]=#0             !STOP
S MW[#1042]=#31            !K
START START
V MW[#1044]
EXIT
```

```
!SIMULATION COMMAND FILE FOR PDP11
!CHARACTER SEARCH
ECHO
RADIX OCTAL
S R[#7]=#1000
S MW[#1000]=#16700      !START:MOV N,R0
S MW[#1002]=#112        !
S MW[#1004]=#166700     !SUB M,R0
S MW[#1006]=#110
S MW[#1010]=#5001       !CLR R1
S MW[#1012]=#5002       !CLR R2
S MW[#1014]=#10103      !LOOP: MOV R1,R3
S MW[#1016]=#20100      !CMP R1,R0
S MW[#1020]=#3001       !BGT T1
S MW[#1022]=#405        !BR CONT
S MW[#1024]=#22702      !T1:CMP #0,R2
S MW[#1026]=#0
S MW[#1030]=#1401       !BEQ T2
S MW[#1032]=#401        !BR CONT
S MW[#1034]=#420        !T2:BR END
S MW[#1036]=#126162     !CONT:CMPB STRING(R1),SUBS(R2)
S MW[#1040]=#1122       !STRING
S MW[#1042]=#1146       !SUBS
S MW[#1044]=#1006       !BNE L1
S MW[#1046]=#5201       !INC R1
S MW[#1050]=#5202       !INC R2
S MW[#1052]=#26702      !CMP M,R2
S MW[#1054]=#42         !M
S MW[#1056]=#1411       !BEQ L2
S MW[#1060]=#403        !BR L3
S MW[#1062]=#5203       !L1:INC R3
S MW[#1064]=#10301      !MOV R3,R1
S MW[#1066]=#5002       !CLR R2
S MW[#1070]=#20167      !L3:CMP R1,N
S MW[#1072]=#22         !N
S MW[#1074]=#3747       !BLE LOOP
S MW[#1076]=#5005       !END:CLR R5
S MW[#1100]=#0          !STOP
S MW[#1102]=#166701     !L2:SUB M,R1
S MW[#1104]=#12         !M
S MW[#1106]=#62701      !ADD #STRING,R1
S MW[#1110]=#1122       !STRING
S MW[#1112]=#10105      !MOV R1,R5
S MW[#1114]=#0          !STOP
S MW[#1116]=#24         !N
S MW[#1120]=#12         !M
S MW[#1122]=#41101      !STRING
S MW[#1124]=#42103
S MW[#1126]=#43105
S MW[#1130]=#44107
S MW[#1132]=#45111
S MW[#1134]=#46113
S MW[#1136]=#47115
S MW[#1140]=#50117
```

```
S MW[#1142]=#51121
S MW[#1144]=#52123
S MW[#1146]=#43105
S MW[#1150]=#44107
S MW[#1152]=#45111
S MW[#1154]=#46113
S MW[#1156]=#47115
START START
V R[#0:#7]
EXIT
```

APPENDIX C

Simulation Command File

to Detect

DFAULT in AF85

```
!SIMULATION COMMAND FILE FOR AF85
!DIAGNOSTIC PROGRAM TO DETECT DFAULT
!IN MEMORY
ECHO
S PC=#3000
S M[#3000]=#100400       !CLR R1
S M[#3001]=#14021        !TST:MOV #10,A(R1)
S M[#3002]=#10
S M[#3003]=#2000
S M[#3004]=#15052        !MOV A(R1),R2
S M[#3005]=#2000
S M[#3006]=#4012         !CMP #10,R2
S M[#3007]=#10
S M[#3010]=#161200       !BNE FAULT
S M[#3011]=#10           !OFFSET
S M[#3012]=#11           !ADD #1,R1
S M[#3013]=#1
S M[#3014]=#4011         !CMP #100,R1
S M[#3015]=#100
S M[#3016]=#143200       !BEQ END
S M[#3017]=#2
S M[#3020]=#141200       !BR TST
S M[#3021]=#177760
S M[#3022]=#60000        !END:STOP
S M[#3023]=#11           !FAULT:ADD #A,R1
S M[#3024]=#2000
S M[#3025]=#60000        !STOP
EXIT
```