11-20-1985

AMISS: a microprogramming simulation system for educational use

Robert Pesar

ROCHESTER INSTITUTE OF TECHNOLOGY
College of Applied Science and Technology
Department of Computer Science

A thesis presented to the faculty of
Rochester Institute of Technology
in partial fulfillment
of the requirements for the
Master of Computer Science

AMISS

A microprogramming simulation system for educational use.

by

Robert J. Pesar

APPROVALS

THESIS ADVISOR:

John L. Ellis

John Ellis, Ph.D.

COMMITTEE MEMBERS:

Chris Comte

Chris Comte

Jeffrey Lasky

Jeff Lasky

I am to be contacted each time a request for reproduction of this document is made. I can be reached at the following address:

November 20, 1985

Robert J. Pesar

Robert J. Pesar

ACKNOWLEDGMENTS

I wish to thank the faculty members in the Department of Computer Science and the Department of Computer Engineering at Rochester Institute of Technology whose assistance was invaluable in the completion of this project.

The Department of Computer Science:

Professor Chris Comte

Professor John Ellis, Ph.D.

Professor Jeff Lasky

Professor Margaret Peek

The Department of Computer Engineering:

Professor Tony Chang, Ph.D

Professor Roy Czernikowski, Ph.D.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Microprogramming is specified as a topic of study in both the ACM Curriculum for the B.S. in computer science [1] and the IEEE Curriculum for undergraduate computer engineering [2]. Generally, it is included as part of a course in computer architecture at the undergraduate and graduate level.

Among the texts commonly used in computer architecture are four that present the topic of microprogramming in a similar manner [3,4,5,6]. Each presents a hypothetical microarchitecture and a microprogram written in a high level microprogramming language that is capable of implementing all or some part of an instructions set. From discussion with faculty members at Rochester Institute of Technology who are involved in teaching computer architecture at the undergraduate and graduate level, it was suggested that a microprogramming simulation system might be a useful addition to the teaching of microprogramming principles. Such systems are in fact in use at a few universities in the United States and Canada. [7,8,9]. Each of these systems presents a hypothetical microarchitecture, a high level microprogramming language, and a microassembler. A typical assignment for the student is to implement a small instruction set or to add some new instructions to the instruction set that the professor has already provided. This task is accomplished through the use of some interactive debugging tool. While apparently useful and well received, these systems are inflexible in that both their microarchitecture and the syntax and semantics of their high level microprogramming language are fixed. This inflexibility could limit a professor's ability to tailor the elements of the hypothetical microarchitecture to the needs of a particular group of students. In addition, it is desirable to have a simulator available that matches the microarchitecture described in the textbook that is being used in the course. A fixed system disallows this.

With the above discussion in mind, AMISS (A Microprogramming Simulation System) was developed. In the development of AMISS, key attributes for a useful educational

microprogramming system were identified. These are:

1. That the system be flexible in its ability to allow the design of the microarchitecture and the microprogramming language, and that these two parts easily combine into a complete simulated system.

2. That the system be flexible in allowing the professor to present to the students a completely designed architecture and instruction set interpreter, just architecture specifications and a instruction set, or anything in between.

3. That one user interface to the executing simulator be from a debugger that is powerful enough to trace and affect execution but not overwhelming in its command repetoire.

Kernighan feels that many large programming projects can best be approached as language development tasks.[11] Among the program development tools available under the UNiX operating system is YACC [10]. YACC is a parser generator that accepts an LALR(1) grammar and produces a program that is capable of recognizing the context of words and acting accordingly. Since AMISS required a good deal of language development, YACC was used as the tool upon which it is based. Further, since YACC is itself a C language program and requires C programs to work, AMISS is written in C.

A primary goal in developing AMISS was to identify those details common to the description of microarchitectures and to combine those details into a Hardware Description Language (HDL). Rather than developing a new programming language, AMISS's HDL is based on the C programming language and an HDL program is translated into a C program. An AMISS HDL program is divided into two parts. In the first part the specifications of the elements of the architecture and of the control memory format are identified. In the second part, the course of events within the processor governed by the contents of a control memory word are described.

Another goal of AMISS was to provide the

means whereby a designer could build a microassembler to work in tandem with the microprogrammable simulator. While YACC is extremely useful for writing assemblers, it requires some effort to learn. AMISS assist the designer in writing a microassembler by providing an interface to a YACC based general microassembler. That interface is named MPL. If MPL is given an HDL program and a listing of the statements of the microlanguage and the actions to be taken when the statements are recognized, it produces a microassembler whose output can be used by the hardware simulator.

Since AMISS is intended to be used as an educational tool, microarchitectures described in textbooks serve as test cases.

The remainder of this document contains a detailed discussion of AMISS. Chapter 2 describes the elements of the system and what fuction each element performs. Chapter 3 contains a formal description of the syntax of the HDL with examples of HDL statements. Chapter 4 contains a formal description of the syntax of the YACC interface MPL and examples of MPL statements. Chapter 5 presents two complete test cases derived from a textbook example and an example from the literature. One of these uses a vertical control word format [7] and the other a horizontal format [8]. Appendix 1 contains a list of HDL and MPL reserved words. Appendix 2 contains the syntax description based on the Backus-Naur Form (BNF). Appendix 3 contains a complete, step by step example of a system designed and simulated using AMISS. The system is based on one described by Tannenbaum [6]. Appendix 4 contains a description of the commands available in the debugger interface. Appendix 5 contains suggestions for using the various parts of AMISS effectively. Appendix 6 contains a guide for running AMISS. Appendix 7 contains an explanation of AMISS error reporting statements.

CHAPTER 2: SYSTEM DESCRIPTION

The AMISS system consists of two broad sections that may be considered as distinct from one another. These are the HARDWARE section and the MICROASSEMBLER section. The HARDWARE section provides the user with the capability of defining a microarchitecture and producing from the definition a program which simulates the microarchitecture. The MICROASSEMBLER section provides the user with the capability of defining a microprogramming language and producing from the definition a microassembler that is able to assemble a program written in the microprogramming language into a simulated control memory. Although it is possible to use each section separately, the two sections are designed to work with each other and together to provide an effective system for microprogrammable processor simulation. Figure 2.1 is a a diagram of the AMISS system.

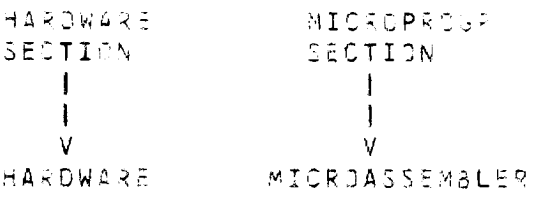


FIGURE 2.1. General diagram of AMISS system.

The design of AMISS relies heavily on the UNIX utility program YACC [12]. YACC is a parser generator that accepts an LALR(1) grammar and produces a program that is capable of recognizing the context of words and acting accordingly. AMISS is designed to accept user input and to produce from the input one executable program that can simulate hardware and and another that can assemble microcode. To do this, AMISS builds files containing C language programs and compiles the programs. All the programs that make up AMISS are written in C. In this chapter, the HARDWARE and MICROPROGRAMMING sections will be considered separately. The hardware simulator produced by the HARDWARE section has options available to it for execution. One of these options is a debugging option. The final part of this chapter

discusses the debugger.

2.1 THE HARDWARE SECTION

Figure 2.2 is a diagram of the hardware section. HRUN is a program that is able to accept a hardware description of a microprogrammable processor. HRUN is produced by the compilation of a number of system files. Among these is a YACC based file that is the parser for the hardware description file. The hardware description file contains a program that is supplied by the user and written in a hardware description language. The program is divided into two sections. The first section is the Specifications section in which the elements of the processor, the control word format(s), the memory, any initializations, and any varibles and user defined files are defined. The second section is the Register Transfers Section. This section contains a program written in HDL, a langauge that is a subset of the C programming language. This program is a description of the register transfers that occur in the processor dependent upon the contents of the fields of a word in simulated control memory. The syntax of the hardware description langauge is contained in Appendix 2.

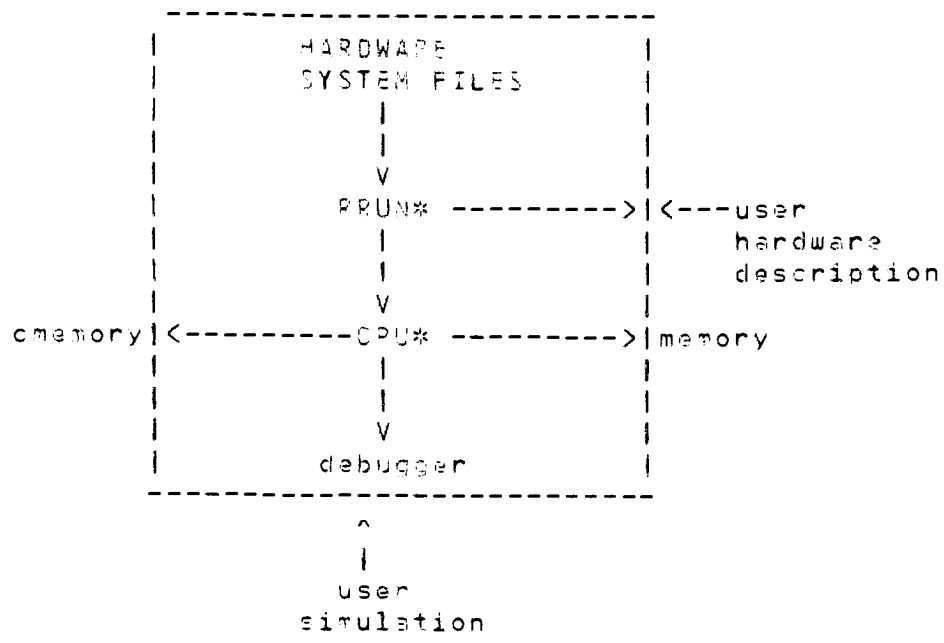


Figure 2.2. Hardware Section diagram showing system files, executable programs (*), and user interfaces.

A key to the functioning of the simulator is the use of flow of control statements with the AMISS keyword SUBCYCLE. The REGISTER TRANSFERS program must break up a single hardware cycle into one or more subcycles. It also must provide the code to perform register transfers depending on the value of the fields of a control memory word pointed to by the value of a microprogram counter that is designated by the AMISS keyword MPC. It must assemble the register transfers that are valid in a particular subcycle into a group and ensure that they are executed only when they are supposed to be. This can be accomplished by enclosing the register transfers for one subcycle inside an IF (SUBCYCLE EQ x) statement. Also, the REGISTER TRANSFERS program must provide for obtaining another control word by changing the value of the MPC. Finally, if more than one control word format is used, the REGISTER TRANSFERS section must refer to each format by the keyword OPCODE and group the register transfers under a statement that will allow them to occur only if the control word in question is of the apporiate format.

When RRUN is invoked with no options,

```
$RRUN file
```

it takes the user hardware description and produces a number of system files. The content of some of these files is invariable, while others are dependent upon the hardware description file. These files are automatically compiled using the UNIX make faciltiy into the executable hardware simulator, CPU. Then they are placed in a directory named RRdir. It is up to the user to remove them. RRUN has one option, -h.

```
$RRUN -h file
```

This option is used if it is necessary to resubmit a hardware description to RRUN that varies from the last submitted hardware description only in the register transfer section (ie. NOT in the specifications section). The purpose of the -h option is to save the time of doing a complete recompilation. Thus, if one anticipates using the -h option, the files in RRdir should be saved.

The executable simulator is named CPU. To perform a simulation, CPU requires 2 files: cmemory and memory. Cmemory is a file of decimal integers separated by white space representing control memory. Control memory is interpreted by CPU to be a microprogram that is able to implement the machine code represention of a program written in some assembly language. CPU interprets the contents of the file memory to be the hex representation of an assembly language program. Each integer in memory represents the contents of one main memory address. If present, CPU uses a third file named instrfile. CPU expects instrfile to contain a condensed form of a high level language microprogram. The two files memory and cmemory must be provided by the user. File memory must be created using an editor. File cmemory may be created using an editor or it may be written by the MICROPROGRAMMING section of AMISS. File instrfile must be provided by the the MICROPROGRAMMING section of AMISS. CPU has these options:

```
$CPU    run the simulator, stop when
        processor decodes a macro halt
```

instruction; report that processor halted;

CPU -n run the simulator, providing a heading containing the contents of each control word, and a count of the number of control instructions executed, report that processor halted;

$CPU -d enter a debugging program that provides runs the CPU simulation according to commands issued by the user at a terminal.

2.2 THE MICROPROGRAMMING SECTION

Figure 2.3 is a diagram of the Microprogramming section.

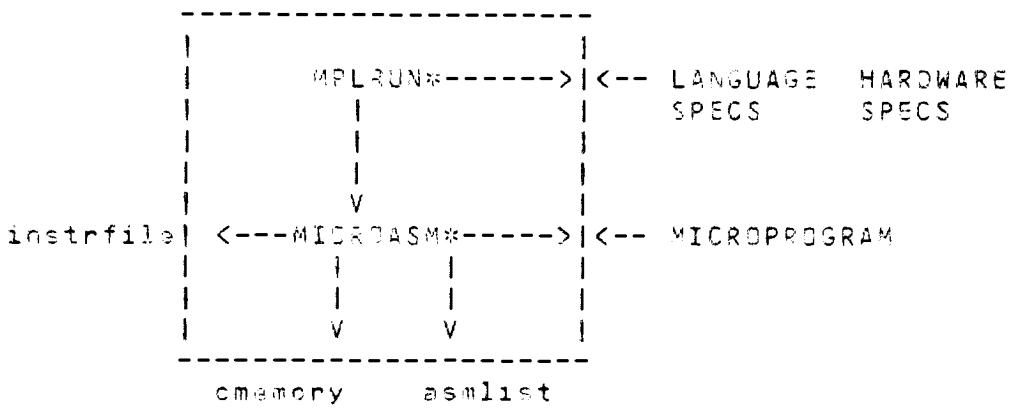


Figure 2.3. Diagram of the microprogramming section. Executable files marked *.

Referring to Figure 2.3, MPLRUN is an executable file. One of the files from which MPLRUN is derived is a YACC based program that allows MPLRUN to accept two input files. One of the files is an HDL hardware description file (HARDWARE SPECS in Figure 2.3). The other is a file containing a description of the grammar rules and the actions to be carried out when one of the grammar rules is recognized (LANGUAGE SPECS in Figure 2.3). MPLRUN uses the two input files to produce a number of system files. One of these files

is a YACC based assembler that incorporates the user language specifications into a generalized assembler. When this file is compiled with the rest of the system files, a microassembler is produced (MICROASM in Figure 2.3). The system files are placed in a directory named MWdir. It is up to the user to remove them. The microassembler is able to parse a microprogram. If the microprogram obeys the syntax rules of the languge as specified by the LANGUAGE SPECS and the the rules of the generalized assembler, three files are produced. The first is cmemory, a file containing a simulated control memory that has been assembled according to the actions specified in LANGUAGE SPECS. The second is instrfile, a file that contains a form of the microprogram that can be used by the debugger portion of CPU for reporting instructions. The third file is of greatest interest to the user. This is the file asmlist. It is a microassembler listing file containing the compiled control memory word assembled for each instruction in the microprogram along with the mnemonic form of the microprogram.

If language specifications reside in a file called LANGUAGE and HARDWARE is an HDL hardware specification, MPLRUN is executed by:

```
%MPLRUN LANGUAGE HARDWARE
```

MPLRUN assumes that the program HARDWARE is syntactically correct (ie. has already been accepted by RRUN). If LANGUAGE is syntactically correct, MPLRUN will automatically produce an executable program called MICROASM which is the microassembler. If LANGUAGE contains a syntax error, MPLRUN will report the line number on which the error occurred and the token which caused the error. If MCODE is a microprogram, it is assembled by:

```
%MICROASM MCODE
```

If MCODE meets the rules of the microlanguage, files asmlist cmemory and instrfile are produced.

2.3 The CPU DEBUGGER

As discussed in section 2.1, the

hardware simulator CPU may be invoked with a debugger option:

```
3CPU -d
```

This option affords the abiltiy to exert control over a hardware simulation. The debugger consists of a small command language whose statements are input interactively from a terminal.

The debugger operates at two distinct states. From the standpoint of hardware simulation, state 1 is a non-executing state. From state one, any command that is part of the language may be input. The user is prompted from state one by:

```
ENTER1:
```

signaling that the debugger is waiting for user input. A command is entered followed by a carriage return. If the command is valid, it is executed by the debugger and another ENTER1: prompt is issued. All commands of the debugger are valid from state one.

State 2 is of the debugger entered by issuing the command that causes the hardware simulation to begin: run. In state 2, a subset of the entire command language is available. The hardware simulation will proceed and information concerning the state of the simulation will be presented on the terminal screen. The nature of this information will depend upon the command that caused the simulation to initiate. At points in state 2 that correspond to the beginning of processor SUBCYCLES, the user will be prompted by:

```
ENTER2:
```

Here, the debugger requires that a user command be input before the simulation can continue. After the number of hardware cycles that have been specified by the state 1 command have been executed, the debugger returns to state one. Here, the entire process starts over.

The manual for the debugger command language is in Appendix 4.

CHAPTER 3: THE SYNTAX OF HDL

The user program that is submitted to RRUN is written in a hardware description language called HDL. RRUN translates the HDL program into a number of C programs that are submitted to the C compiler to produce an executable hardware simulator named CPU. The syntax rules of a HDL program are in Appendix 2. In this chapter, the rules will be considered in detail.

An HDL program consists of begin tokens followed by a hardware specification followed by end tokens.

```
program ::=  list

list ::=
     |list   @ @     BEGIN   @
        define_Cfunct        define_formats
          define_parts        define_memory
            define_inits     define_rts
                   @   @
```

An HDL program may be divided into two parts, the SPECIFICATIONS part and the REGISTER TRANSFER part. In the rule "list" above, the SPECIFICATIONS part is made up of define_Cfunct, define_formats, define_parts, define_memory, and define_inits. The REGISTER TRANSFER part is made up of the define_rts. Since HDL is a subset of C, the syntax of the constructs of the REGISTER TRANSFERS part is identical to C, except that HDL accepts only uppercase letters. In the discussion of the syntax of HDL, the two parts will be considered separately.

3.1 SPECIFICATIONS

3.11 CFUNCTIONS SPECIFICATIONS

The CFUNCTIONS specifications is optional and any of the four statements that make it up are optional. The statements that are used must be listed in the order MAKEFILES DEFINES INCLUDES VARS. Each of the four statemtents begin with "<" followed by one of the keywords MAKEFILES DEFINES INCLUDES and VARS. Each statement is terminated by ">". No syntax checking is made after the keyword and before the final ">".

The correct way to use each section is:

MAKEFILES list any user files that are to be included in the makefile that builds the hardware simulator as

file.o file.o ...

the complete list must fit on one line; these files can contain valid C functions that may be called from within the HDL program; to reference any partnames of the HDL program, include the AMISS system file vardefines.h as #include "var_defines.h"

DEFINES list any user defined names as

#define name1 name2

#define name3 name4 ...

INCLUDES list any file names to be included in any files named in the makefiles section as

#include myfile1.h

#include myfile2.h

VARS list any user defined variables according to the rules for declaring C variables

The syntax rules of the define_Cfunct section are:

```
define_Cfunct::=
            |    CFUNCTIONS   Clist  @

Clist ::=
         |makefiles
         |defines
         |defines makefiles
         |includes
         |includes        makefiles
         |includes defines
         |includes defines makefiles
         |vars
         |vars     makefiles
         |vars     defines
         |vars     defines makefiles
         |vars     includes
         |vars     includes         makefiles
         |vars    includes defines
         |vars    includes defines makefiles

vars ::=          <  VARS       anything    >

includes ::=      <  INCLUDES    anything    >

defines ::=       <  DEFINES     anything     >

makefiles ::=     <  MAKEFILES  anything     >
```

"anything" is a lexical entity consisting of any ascii character or series of characters

An example using all of the CFUNCTIONS options is:

```
         @ CFUNCTIONS
         < VARS
                 int  temp1;
                 short temp2;   >
         < INCLUDES
                 #include    "myfile.h"  >
         < DEFINES
                 #define     alu1    aluport1 >
         < MAKEFILES
                 inout.o output.o          >
```

3.11 FORMAT SPECIFICATIONS

In the format section, the format definitions are listed. mDL keyword FORMATS is followed by one of FORMAT1, FORMAT2, FORMAT3, or FORMAT4. Each is followed by

":". Next, the width of each field in that format is listed within the tokens "<" and ">" followed by "=" followed by the name of the field. Fields within a single format are separated by "," and a format specification is terminated by ";". The first field of each format must be specified as having width 2 and name OPCODE. The syntax rules of the format section are:

```
define_formats::= FORMATS     format_descr   @

format_descr::= formatnum  :  <   NUM    >   =
                              OPCODE  ,  format_specs  ;

formatnum::=    FORMAT1 | FORMAT2 | FORMAT3 | FORMAT4

format_specs::= <   NUM   >   =    expr
     | format_specs  ,   <   NUM   >     =    expr
```

An example of a format specifications is:

```
@ FORMATS
        FORMAT1:        <2> = OPCODE,
                        <4> = ACNTRL;

        FORMAT2:        <2> = OPCODE,
                        <8> = ADDRF;
```

Consult the reserved word list in Appendix 1 or section 3.21 for words not to be used as format field names.

3.13 PARTS SPECIFICATIONS

In this section all the elements of the microarchitecture are listed after a specification of the width of the parts. Parts having the same width may be listed as a group separated by ",". A single group is terminated by a ";". One of the parts listed must be named MPC. The compiler does not enforce this, but not doing so will result in a warning. In the following syntax rules for the parts specification, the non-terminal "expr" is reduced in a subsequent rule:

```
define_parts ::=   PARTS  parts_list  @
      | parts_list < NUM  >  =  name_list :

name_list ::=  expr
        | name_list   ,    expr
```

An example parts specification is:

```
@ PARTS          <16> = R0,   R1,   PC;
    <8> = MPC;
```

Consult the reserved word list in Appendix 1 or section 3.21 for words not to be used as part names.

3.14 MEMORY SPECIFICATIONS

In this section the two keywords MEM and CMEM are used to specify the number of words and the width of memory and control memory. In the syntax below, NUM is any decimal integer. Each NUM is surrounded by "<" ">". For each of MEM and CMEM, the first number is the number of words and the second is the width. Either MEM or CMEM may be listed first. The syntax rules are:

```
define_memory ::=  MEMORY  mem_specs  @

mem_specs ::=  <  NUM  >  <  NUM  >  =  memname  ;
             ;    memname    =    <     NUM     >
                  <    NUM    >

memname ::=     MEM
          |     CMEM
```

An example memory specification is:

```
@ MEMORY
     <128><16> = CMEM;
     <1024><16> = MEM;
```

For AMISS to work properly, the width of the CMEM specification must match the sum of the field widths of the widest control format listed in the format specification.

3.15 INITIALIZATIONS

In this section any part named in the part section may be initialized to a constant. The constant is interpreted as a hexidecimal integer. The syntax rules are:

```
def_inits
        |  INITIALIZATIONS   init_list     ;

init_list       id  =  NUM ;
        |       id  = id [  NUM  ] ;
        |       init_list id = NUM ;
        |       init_list id = id [ NUM ]
```

NUM is a lexical entity that is made up of hexidecimal digits 0 1 2 3 4 5 6 7 8 9 a b c d e f.
A hexidecimal constant may be written either as a series of hexidecimal digits or a 0x followed by a a series of hexidecimal digits.

3.2 The REGISTER TRANSFERS

The body of the simulator is described in the REGISTER TRANSFERS section of an HDL program. The constructs of this section are derived from the C programming language constructs that deal with arithmetic operations, assignment, and flow of control. The reader may wish to consult a reference describing the C programming language for additional discussion of these constructs. [12] [13]

3.21 IDENTIFIERS

An identifier is a lexical entity composed of an uppercase letter followed by one or more uppercase letters or digits. Reserved words may not be redefined and therefore are not valid identifiers. The following list contains the reserved words of HDL. The reserved words of the MICROPROGRAMMING section are also included. While these are acceptable identifier names in HDL, they should be avoided because they will not be accepted as identifiers in the MICROPROGRAMMING section.

BEGIN	BOR	INITIALIZATIONS
NUM	IF	ELSE
SWITCH	CASE	DEFAULT
BREAK	RETURN	EQ
NE	GT	GE
LT	LE	
OR	BOR	NOT
SL	SR	DIV
ID	SELECT1	SELECT2
SELECT3	SELECT4	DEFINE
CONSTANTS	MICROENGINE	FORMATS
FORMAT1	FORMAT2	FORMAT3
MEM	CMEM	CFUNCTIONS
VARS	INCLUDES	MAKEFILES
DEFINES	OPCODE	THEN
GOTO	SEMANTICS	SELECT
TEST	LSHIFT	RSHIFT
INV	BAND	RD
WR	CONT	CONTFORMAT

3.22 COMMENTS

Comments are arbitray strings placed between the delimiters /* and */. Comments are valid anywhere within an HOL program after the BEGIN 00. An example comment is:

```
/**********************
*     here    is    a    comment    *
**********************/
```

3.23 ARITHMETIC OPERATORS

The binary arithmetic operators

+ - * DIV AND OR

correspond to the mathematical operations of addition, subtraction, multiplication, division, bitwise and, and bitwise or. The unary operator NOT corresponds to negation. The assignment operators are:

= SR SL

SR stands for SHIFTRIGHT, SL for SHIFTLEFT. SR is the C operator >>=, SL is <<=. They are a combination of binary operators and assignment statements. For example, TEMP SR 1 shifts the contents of TEMP right 1 bit and assigns the value to variable TEMP. The rules of precedence and associativity for the

arithmetic and the assignment operators are found in section 3.24.

3.24 RELATIONAL OPERATORS

The following table contains the operators that are used to affect flow of control:

Relational, equality, and logical operators			C language equivalent
relational operators:	less than	LT	<
	greater than	GT	>
	less than or equal to	LE	<=
	greater than or equal to	GE	>=
equality operators:	equal	EQ	==
	not equal to	NE	!=
logical operators:	negation	NOT	!
	logical and	BAND	&&
	logical or	BOR	||

Operators have the following rules of precedence and associativity:

OPERATOR	ASSOCIATIVITY
unary -	right to left
* DIV	left to right
+ -	left to right
LT LE GT GE	left to right
EQ NE	left to right
AND BAND	left to right
OR BOR	left to right
= SS SL	right to left

All the relational, equality and logical operators other than unary - are binary. They operate on expressions and yield either the value of 0 or 1. False is represented by 0 and true is represented by any nonzero value.

3.25 MICROENGINE SYNTAX

A register transfer description begins with the keyword MICROENGINE followed by the begin token "{" followed by a listing of the register transfers followed by the end token "}".

```
define_rts ::= MICROENGINE { rts_list }
```

The rts_list is composed of one or more of statements. Each statement may be preceded by a label composed of a valid id followed by a ":".

```
rts_list ::=    st
          |     label          st
          |     rts_list       st
          |     rts_list       label          st

label ::=   id  :
```

The various forms of a statement (st) make up the core of a register transfer description. Explanations for the alternatives for the statement will be considered one at a time in the remainder of this section. The syntax rules for a statement are:

```
st ::=  ;
     |  {   }
     |  expr    ;
     |  compound_st
     |  IF  (   expr    )    st   else_st
     |  SWITCH    (    expr    )  {  case_list }
     |  select    (    selectlist    )     ;
```

A statement may be empty of the form

```
        ;
    or
        { }
```

Alternatively, a statement may be composed of an expression An expression may be empty. Alternatively, an expression may consist of an primary, a - expr, or an expression followed by any aritmetic, relational, assignment, or equality operator followed by an expression.

```
expr ::=
         |  primary
         |  expr binop expr
         |  expr asignop expr
         |  unop expr
```

Some examples are:

```
REG1 = 10
REG1 = REG2
CMPL = - ABUS
ACNTRL LE 10
SHIFTER SR 1
```

A primary may be an ID, a NUM, a parenthsized expression, a primary followed by a left parenthesis followed by an expr followed by a right parenthesis, or an expr followed by a left bracket followed by an expr followed by a right bracket. The syntax rules for a primary are:

```
primary ::=     id
          |     const_expr
          |     ( expr )
          |     primary [ expr ]
          |     primary ( expr )
```

Some examples are

```
A
0xfff
29
MEM[MAR]
OUTPUT()
```

The syntax rules for the binop, unop, and asignop are:

```
binop ::= EQ | GT | GE | LT | LE |
          NE | AND | OR | EOR |
          &  | *  | + | - | DIV |
          SL  | SR | BAND | BOR

unop ::= NOT

assignop ::= =
```

A const_expr is a number, either decimal or hexidecimal.

```
ABUS = 0xfff;  ACNTRL = 10;
```

A compound statement is a series of statements surrounded by braces. The main use of the compound statement is to group statements into an executable unit. Wherever it is syntactically correct to place a statement, it is correct to place a compound statement.

The syntax rules for a compound statement are:

```
compound_st ::=  { st_list }

st_list ::=      st
           |     st stlist
```

An example of a statement is:

```
        {
                ACNTRL = 10;

                {
                    BCNTRL = 10;
                    CNTRL = 20;
                }
        }
```

Grouping of statements is used to achieved the desired flow of control in such constructs as the IF STATEMENT.

The syntax rules for the IF STATEMENT are:

```
if_st   ::=  IF ( expr ) st else_st

else_st ::=
        | ELSE st
```

In a construction of the form

```
        IF ( expression )
                statement1
        next statement
```

If expression is nonzero (true), then statement1 is executued; otherwise statement1 is skipped and control passes to the next statement. An example of a simple IF statement is:

```
        IF ( ACNTRL EQ 2)
                ABUS = R2;
```

The if_else statement is of the form

```
        IF ( expression )
                statement1
        ELSE
                statement2
        next statement
```

If expression is nonzero then statement1 is executed and statement2 is skipped; if expression is zero, then statement1 is skipped and statement2 is executed. In either case, control next passes to next statement. An example of the if_else construct is:

```
IF ( C EQ 1 )
        IF ( N EQ 0 ) MPC = ADDRF;
        ELSE MPC = MPC + 1;
```

An ELSE attaches to the nearest IF. Therefore in the above example the ELSE is the alternative to the "IF (N EQ 0)" and not the "IF (C EQ 1)". If the example were written:

```
IF ( C EQ 1 ) {
        IF ( N EQ 0 ) MPC = ADDRF;
}
ELSE MPC =  MPC + 1;
```

the ELSE would attach itself to the "IF (C EQ 1) ". The SWITCH statement is a multi-way conditional statement generalizing the if_else statement. The syntax rules for the SWITCH statement are:

```
    |  SWITCH ( expr ) {   case_list   }

case_list ::=  CASE NUM : expr BREAK;
     |         DEFAULT : expr BREAK;
     |         case_list CASE NUM : expr BREAK;
     |         case_list DEFAULT : expr BREAK;
```

The integral expression following the SWITCH is evaluated. Control then jumps to the appropriate case label and the expression at that label is executed. The DEFAULT case may be included, generally as the last case listed. If it is included, and if no other cases are appropriate, control will pass to it. An example of a case statement is:

```
SWITCH ( ACNTRL ) {
        CASE 0: ABUS = R0; BREAK;
        CASE 1: ABUS = R1; BREAK;
        CASE 2: ABUS = R2; BREAK;
        DEFAULT: ABUS = ABUS; BREAK;
}
```

The select statement provides a shortened form for writing a switch statement. It is

not a part of C, but is especially suited to writing microprogrammable simulators. It is rewritten by RRUN into a switch statement. The syntax rules for a select statement are:

```
|    select  (  selectlist  )  ;

select ::=      SELECT1
                SELECT2
                SELECT3
                SELECT4

selectlist ::=  primary    ,    primary    ,    idlist

idlist ::=      NUM        :    primary
                idlist     ,    primary
```

In the rule idlist, NUM is a constant such that 0 < NUM <= 25. NUM must match the number of primaries listed in the rule idlist. In the rule select, SELECT1 and SELECT2 assist in writing vertically encoded simulators. SELECT3 and SELECT4 assist in writing horizontally encoded simulators. The following are examples of each of the SELECT statements:

```
SELECT1( ABUS, ACNTRL, 4:
        R0, R1, R2, PC ) ;

Rewritten into:

SWITCH ( ACNTRL ) {
        CASE 0: R0 = ABUS; BREAK;
        CASE 1: R1 = ABUS; BREAK;
        CASE 2: R2 = ABUS; BREAK;
        CASE 3: PC = ABUS; BREAK;
}
SELECT2( ABUS, ACNTRL, 4:
        R0, R1, R2, PC) ;

Rewritten into:

SWITCH ( ACNTRL ) {
        CASE 0: ABUS = R0; BREAK;
        CASE 1: ABUS = R1; BREAK;
        CASE 2: ABUS = R2; BREAK;
        CASE 3: ABUS = PC; BREAK;
}
```

```
SELECT3 ( ABUS, ACNTRL 6:
        R0, R1, R2, IR, PC, MBR) :

Rewritten into:

SWITCH ( ACNTRL ) {
        CASE 1: ABUS = R0; BREAK;
        CASE 2: ABUS = R1; BREAK;
        CASE 4: ABUS = R2; BREAK;
        CASE 8: ABUS = IR; BREAK;
        CASE 16: ABUS = PC; BREAK;
        CASE 32: ABUS = MBR; BREAK;
}
SELECT4 ( ABUS, ACNTRL, 6:
        R0, R1, R2, IR, PC, MBR) ;

Rewritten into:

SWITCH ( ACNTRL ) {
        CASE 1: R0 = ABUS; BREAK;
        CASE 2: R1 = ABUS; BREAK;
        CASE 4: R2 = ABUS; BREAK;
        CASE 8: IR = ABUS; BREAK;
        CASE 16: PC = ABUS; BREAK;
        CASE 32: MBR = ABUS; BREAK;
}
```

CHAPTER 4: THE SYNTAX OF MPL

This chapter describes the syntax rules of the microprogramming language input to the microassembler generator named MPL. It is assumed that an HDL hardware program exists that has been accepted by the hardware simulation generator RRUN and that MPL is being used to write a microassembler for a microlanguage that is to be used in conjunction with the simulator. If the language description is in file LANGUAGE and the HDL hardware description is in file HARDWARE, MPL is executed by:

$MPL LANGUAGE HARDWARE

One of the components of MPL is a YACC based parser that accepts a program containing a user definition of the syntax rules and associated actions of a microprogramming language. MPL uses that program and a hardware specification written in HDL and produces a microassembler that is able to parse a program written in the microprogramming language. If the program is accepted, MPL creates three files named cmemory, asmlist, and instrfile. File cmemory contains decimal integers that represent a control memory assembled according to the actions carried out upon recognition of statements of the microprogramming language. File cmemory is intended to provide the control memory file that is required by CPU, the hardware simulator produced by RRUN. File asmlist is a listing file containing the cmemory, a listing of the user microprogram, and a table containing the addresses of labels of the microprogram. File instrfile is a system file that contains a condensed form of the microprogram that is used by the debugger part of the hardware simulator.

An MPL program may be logically divided into two sections. In the first section, the grammar and actions that define the TEST portion of the microprogramming language are described. In the second section, the grammar and actions that define the REGISTER TRANSFER statements of the language are described.

The syntax rules for an MPL program are:

```
program ::=     list

list ::=
        |       list define_test nt
```

We will consider TEST and REGISTER TRANSFER sections separately.

4.1 The TEST DEFINITION

In the TEST section the input to MPL must define the rules and actions for the TEST portion of the microprogramming language. The syntax rule for this section is:

```
define_test::=  DEFINE TEST { %% MMTEST :
                                  %% }
```

The reserved words DEFINE TEST and the token "{" are followed by the definition. MPL does no syntax checking of the TEST specification after the "%% MMTEST : " and until the final %%. Anything is acceptable to MPL, however. the TEST definition must be compatible with the rules of writing a YACC parser and the rules of MPL or the microassembler will not work. These rules are:

1. Any MPL keyword used in the rules that define the TEST is a terminal token in the grammar, i.e. it is not reduced further.

2. Any part named in the parts section of the hardware description is a terminal token.

3. Any user defined non-terminal must begin with a letter and may be followed by one or more letters or digits.

4. Any user defined non-terminal must be completely reduced to terminal tokens within the TEST definition.

5. A user defined non-terminal reduction rule is written by writing the non-terminal name followed by a colon ":". If there are alternatives to reducing the rule, they are separated by a "|". A reduction is terminated by a

semicolon ";".

6. The non alpha characters available for use as terminal tokens are:

, [ ] ( ) =

0 1 2 3 4 5 6 7 8 9

7. Any non-alpha character must be surrounded by quotation marks.

8. A reference to a label in a microprogram is by the keyword LABELOPERAND in a grammar rule.

9. Actions associated with a rule are placed after the rule within the markers "{" and "}".

10. A reference to the address of a label in the actions of a grammar rule is by the keyword MMADDRESS.

11. The keyword OPCODE must be set within the actions of a grammar rule.

The design of MMTEST requires that a combination of the tokens (ie. the MPL reserved words and the architecture parts) and single characters form grammar rule and that actions be associated with the grammar rules. If the microassembler is to be used in conjunction with a hardware simulator produced by RRUN, the actions must be compatible with the control fields described in the specifications parts of the HDL program. The following are examples of TEST specifications:

EXAMPLE 1:

```
DEFINE TEST
{
%%
MMTEST: IF BIT "(" one "," two ")" GOTO LABELOPER
                        {OPCODE = 0;
                         ADDRFIELD = MMADDRESS;
                        }
    |   GOTO LABELOPERAND
                        { OPCODE = 0;
                          ADDRFIELD = MMADDRESS;
                        }
    ;
```

```
one:    IR      { REGISTERFIELD = 0; }
    |   R0      { REGISTERFIELD = 1; }
    |   PC      { REGISTERFIELD = 2; }
    ;

two:   "1" "0"  {BIT_NUMBER = 10;}
    |  "1" "1"  {BIT_NUMBER = 11;}
    |  "1" "5"  {BIT_NUMBER = 15;}
    ;
%%
}
```

A dissection of example 1 with regard to to the rules for defininiing the TEST grammar may be helpful.

Rule 1. MPL keywords used in the grammar are terminals. They are:

IF BIT GOTO LABELOPERAND.

Rule 2. Parts named in the parts section of the hardware description (assume here that the 3 parts IR, R1, and PC were listed) are terminals. They are:

IR, R0, PC.

Rule 3. User defined non-terminals are: one, two.

Rule 4. Non-terminals "one" and "two" are completely reduced to terminals.

Rule 5. Note use of "|" to separate alternatives and ";" to end a rule.

Rule 6. Non-alpha characters used are: (,) 0 1 5.

Rule 7. Note quotation marks around every non-alpha character.

Rules 8.
9.
10.
11. Note use of LABELOPERAND after the keyword GOTO. If "FETCH" were a label in a microprogram, the statement "GOTO FETCH" would be accepted. Note in the actions that ADDRFIELD is set to MMADDRESS. In

the microprogram, the control field ADDRFIELD would be set to the address of label FETCH. Note that OPCODE is set once for each complete reduction.

EXAMPLE 2

```
DEFINE TEST {
%%
MMTEST: IF N GOTO LABELOPERAND  { OPCODE = 0;
                                  ADDRESS = MMADDRESS; }
      | GOTO LABELOPERAND       { OPCODE = 0;
                                  ADDRESS = MMADDRESS; }
      ;
%%
}
```

4.2 The REGISTER TRANSFERS

In the REGISTER TRANSFERS part of the language description specification to MPL, all the valid statements of the language are explicitly described along with their associated actions. The syntax rule that determines whether an attempt to define a statement of the microprogramming language is a valid definition is contained in the MPL grammar rule named "rt".

The rule is:

```
rt ::=      | pname " = " pname semantics
       |    | pname " = " pname op num semantics
       |    | pname " = " pname op " - " num semantics
       |    | pname " = " pname op pname semantics
       |    | RD semantics
       |    | WR semantics
       |    | LSHIFT semantics
       |    | RSHIFT semantics
       |    | pname " = " LSHIFT " ( " pname " ) "
                                          semantics
       |    | pname " = " LSHIFT " ( " pname op pname "" )
                  " semantics
       |    | pname " = " RSHIFT " ( " pname " ) "
                   semantics
       |    | pname " = " RSHIFT " ( " pname op pname " )
                  "  semantics
       |    | pname " = " BAND " ( " pname " , "
                   pname " ) " semantics
       |    | pname " = " BOR " ( " pname " , "
                   pname " ) " semantics
       |    | pname " = " INV " ( " pname " ) "
                   semantics
       |    | pname " = " INV " ( " pname " ) " op pname semantics
       |    |  DEFINE SEMANTICS { const
                  SELECT ( pname , pname , num :
                  nlist )
            }
       |    rt rt
```

while rule rt is quite long, it is not complicated. It simply is a listing of a large number of the possible register transfer and function statements that may occur within the constraints of the keywords of MPL and of the user HDL hardware description (the parts of the hardware description become keywords). Thus, the rule defines how the hardware part names and the MPL keywords may be arranged in order to define the microprogramming language. In rt, pname refers to any parts defined in the HDL hardware description. The non-terminal semantics (reduced later) refers to actions that are to be carried out when a statement is recognized. Looking at the first alternative:

```
| pname " = " pname semantics
```

A statement setting any part of the architecture to any other part followed by a specifications of actions is accepted. For example,

```
| IR "=" MBR { OPCODE = 0; }
```

would become part of the microassembler and would cause " IR = MBR " to become part of the microlanguage. Nonterminal "op" is reduced to

```
op ::= "+" | "-" | AND | OR | EOR | BAND | BOR
```

In the second alternative of rt, statements of the form:

```
PC = PC + 1   { OPCODE = 0; }
```

and

```
ALU = ALU1 AND ALU2 {OPCODE = 0;}
```

would become part of the language if

```
| PC "=" PC "+" "1"  {OPCODE = 0;}
```

and

```
| ALU "=" ALU1 AND ALU2 {OPCODE = 0;}
```

were part of the input to MPL and PC ALU ALU2 were listed as parts in HDL hardware description.

Certain MPL reserved words are intended to be used in specific ways (although they don't have to be). RD is to be associated with a memory read statement and WR with a memory write. LSHIFT and RSHIFT are to be associated with shifting, INV with inverting, BAND with boolean and. Non-terminal num is reduced by the rule:

```
num ::= NUM
      | "NUM"
      | num num
```

NUM is a lexical entity that is any of 0 1 2 3 4 5 6 7 8 9. Non-terminal semantics is reduced by the rules:

```
semantics ::=
          |     { OPCODE  = num ; descr }

descr ::=   id = num ;
          | id [ num ] = num;
          | descr descr
```

In this case, id would refer to a control field name. The second to last alternative in rule rt allows MPL to accept a statement that is designed to provide a short cut way of describing a number of rules and actions to be added to the microassembler of the form "pname = pname { actions } " where the

register transfers differ only in one of the pnames and one of the actions. The syntax rule for this statement is:

```
| | DEFINE SEMANTICS {
        const  SELECT
        ( pname , pname , num : plist ) }
```

Non-terminals are reduced to:

```
   const ::= CONSTANTS : OPCODE = num ;
       |     CONSTANTS : OPCODE = num , clist ;

   clist ::= pname = num , clist

   plist ::= pname
       |     plist , pname
```

The DEFINE SEMANTICS is best explained by example. Take the case of four parts (tokens) named A B C BUS. It is desired to add 3 statements to the microprogramming language such that the semantics associated with each of the statements must set the OPCODE = 0, ENCODE = 1 and the field CONTROL to 0 if BUS = A, to 1 if BUS = B, and to 2 if BUS = C depending on the value of a field called CONTROL. MPL would accept the following:

```
   DEFINE SEMANTICS {
        CONSTANTS: OPCODE = 0, ENCODE = 1;
        SELECT ( BUS, CONTROL, 3:
                 A, B, C )
   }
```

Three statements would then be added to to the microassembler. These would be of the form:

```
   |  BUS = A  {OPCODE = 0; ENCODE = 1;
              CONTROL = 0; }

   |  BUS = B  {OPCODE = 0; ENCODE = 1;
              CONTROL = 1; }

   |  BUS = C   {OPCODE = 0; ENCODE = 1;
              CONTROL = 2; }
```

and the statements

```
        BUS  = A
        BUS  = B
        BUS  = C
```

would become part of the microlanguage. Non-terminal id is reduced by the rule:

```
id ::= ID
```

ID is a lexical entity that matches any MPL reserved word or microarchitecture part name. The complete specification for the grammar of MPL is in Appendix 3.

CHAPTER 5: TEST CASES.

TEST CASE 1.

Parker presents a microprogramming simulation package that is used in teaching computer architecture at the University of Calgary [8]. The hypothetical processor is a 16 bit machine with a horizontally encoded control memory and two control memory formats. The GATE format controls 38 gates and the reading and writing of main memory. The TEST format allows the testing of one bit of any of 8 registers for a 1 or a 0. Figure 1 is a diagram of the processor and the control word formats.

In this simulation, the following changes were made to the control word format:

1) the AMISS field OPCODE takes the place of the opcode bit in both formats, increasing the length of the control word by 1;

2) an ALU function control field ALUF was added to FORMAT1 to control 4 functions, increasing the length of the control word by 2;

The other changes to the processor are:

1) the register called "S" is called "RS" in this simulation;

2) a single cycle is divided into 4 subcycles;

3) two status bits NBIT and ZBIT are added;

Figure 1. Processor diagram and control word format.

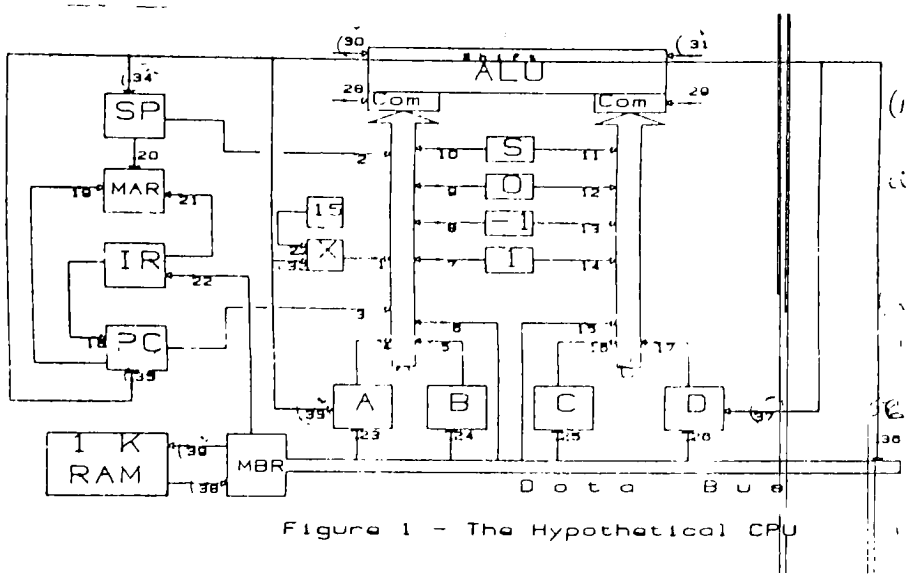
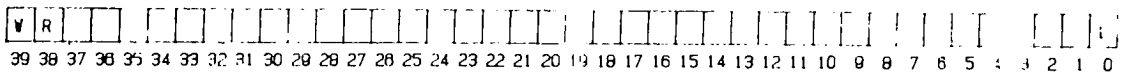


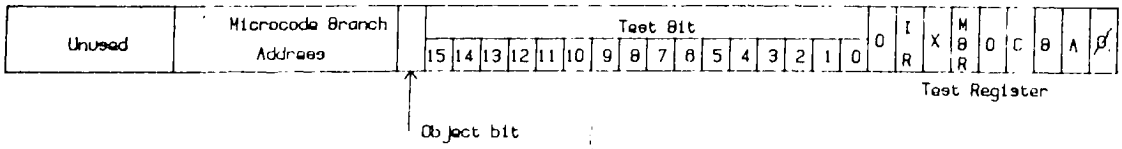
Figure 1 - The Hypothetical CPU

The GATE Instructions :



Each bit of a GATE instruction represents one of the control points (See Fig. 1) A '1' bit allows data to move across a control point. a '0' bit does not. Bit 0 of each microinstruction is the opcode a GATE instruction has an opcode of '1'. Bits 38 and 39 control reading and writing of main memory. respectively.

The TEST Instructions :



The TEST microinstruction allows any bit of certain registers to be tested. In any named field above. only one bit may be non-zero or an error occurs. The action of the TEST instruction is to examine the indicated test register. to see if the specified bit (The TEST field above. 0-15) is equal to the the value of the Object Bit field (1 or 0). If so. a branch is made to the microinstruction specified by the Microcode Branch Address field. If not. then the next instruction in the normal sequence is executed. Opcode for the TEST instruction is '0'.

Figure 2 - The Microinstruction Formats

No other changes were made to the processor or its control format. The listing of the hardware description to describe the processor is:

```
@@ BEGIN
@ CFUNCTIONS
        <VARS   int     temp,temp2; >
@ FORMATS
        FORMAT1:        <2> = OPCODE,
                        <10> = ACNTRL,
                        <7> = BCNTRL,
                        <1> = PCIR,
                        <1> = PCMAR,
                        <1> = SPMAR,
                        <1> = IRMAR,
                        <1> = MBRIR,
                        <4> = REGS1,
                        <1> = XCNTRL,
                        <2> = CMPL,
                        <2> = SHIFTCNTRL,
                        <1> = XFEED,
                        <1> = AFEED,
                        <1> = SPFEED,
                        <1> = PCFEED,
                        <1> = DBUSFEED,
                        <1> = DFEED,
                        <2> = ALUF,
                        <2> = RW;

        FORMAT2:        <2> = OPCODE,
                        <8> = RNAME,
                        <16> = BITNUM,
                        <1> = DEJ,
                        <6> = ADDRF;
@ PARTS
        <16> =  A,B,C,D,X,C15,CZERO,CONE,CMINUSONE,
                SP,IR,PC,PS,
                CMPL1,CMPL2,
                ALU1,ALU2,ALU,
                SHIFTER,
                ABUS,BBUS,DBUS,
                MAR,MBR,MPC;
        <1> =   NBIT,ZBIT;
@ MEMORY
        <128><48> = CMEM;
        <1024><16> = MEM;
@INITIALIZATIONS
        PC = 0;
        CONE = 1;
        CZERO = 0;
        CMINUSONE = 0xffff;
        SP = 0x0014;
```

```
@ MICROENGINE
{
  IF (OPCODE EQ 0) {

        IF (SUBCYCLE EQ 1) {

                DBUS = MBR;

                SELECT3( ABUS,ACNTRL,10:
                        X,SP,PC,A,B,
                        DBUS,CONE,CMINUSONE,CZERO,RS);

                CMPL1 = ABUS;

                SELECT3( BBUS,BCNTRL,7:
                        RS,CZERO,CMINUSONE,CONE,DBUS,C,D);

                CMPL2 = BBUS;

                IF (PCMAR EQ 1) MAR = PC;
                IF (PCIR EQ 1) IR = PC;
                IF (SPMAR EQ 1) MAR = SP;
                IF (IRMAR EQ 1) MAR = IR;
                IF (MBRIR EQ 1) IR = MBR;

        SWITCH (REGS1) {
        CASE 0:  BREAK;
        CASE 1:  A = DBUS; BREAK;
        CASE 2:  B = DBUS; BREAK;
        CASE 3:  A = DBUS; B = DBUS; BREAK;
        CASE 4:  C = DBUS; BREAK;
        CASE 5:  C = DBUS; A = DBUS; BREAK;
        CASE 6:  C = DBUS; B = DBUS; BREAK;
        CASE 7:  C = DBUS; B = DBUS; A = DBUS; BREAK;
        CASE 8:  D = DBUS; BREAK;
        CASE 9:  D = DBUS; A = DBUS; BREAK;
        CASE 10: D = DBUS; B = DBUS; BREAK;
        CASE 11: D = DBUS; B = DBUS; A = DBUS; BREAK;
        CASE 12: D = DBUS; C = DBUS; BREAK;
        CASE 13: D = DBUS; C = DBUS; A = DBUS; BREAK;
        CASE 14: D = DBUS; C = DBUS; B = DBUS; BREAK;
        CASE 15: D = DBUS; C = DBUS;
                 B = DBUS; A = DBUS; BREAK;
        }

                IF (XCNTRL EQ 1) X = C15;

                SWITCH (CMPL) {
                        CASE 0: BREAK;
                        CASE 1: CMPL1 = -CMPL1; BREAK;
                        CASE 2: CMPL2 = -CMPL2; BREAK;
                        CASE 3: CMPL1 = -CMPL1;
                                CMPL2 = -CMPL2; BREAK;
                }
```

```
        ALU1 = CMPL1;
        ALU2 = CMPL2;
}

IF (SUBCYCLE EQ 2) {

        SWITCH (ALUF) {
                CASE 0: ALU = ALU1; BREAK;
                CASE 1: ALU = ALU1 + ALU2; BREAK;
                CASE 2: ALU = ALU1  ALU2; BREAK;
                CASE 3: ALU = -ALU1;BREAK;
        }

        IF ( ALU LT 0 ) NBIT = 1; ZBIT = 0;
        IF ( ALU EQ 0 ) NBIT = 0; ZBIT = 1;
        IF ( ALU GT 0 ) NBIT = 0; ZBIT = 0;

        SHIFTER = ALU;

        SWITCH ( SHIFTCNTRL ) {
                CASE 1: SHIFTER SL 1; BREAK;
                CASE 2: SHIFTER SR 1; BREAK;
                DEFAULT: BREAK;
        }
}

IF (SUBCYCLE EQ 3) {

        IF ( XFEED EQ 1) X = SHIFTER;
        IF ( AFEED EQ 1) A = SHIFTER;
        IF ( SPFEED EQ 1) SP = SHIFTER;
        IF ( PCFEED EQ 1) PC = SHIFTER;
        IF ( DBUSFEED EQ 1) DBUS = SHIFTER;
        IF ( DFEED EQ 1) D = SHIFTER;

        MBR = DBUS;
}

IF (SUBCYCLE EQ 4) {

        SWITCH ( RW ) {
                CASE 1: MBR = MEMEMAR[]; BREAK;
                CASE 2: MEMEMAR[] = MBR; BREAK;
                DEFAULT: BREAK;
        }
        MPC = MPC + 1;
}
RETURN;
}


IF ( OPCODE EQ 1 ) {
        IF ( RNAME EQ 126 ) {
                IF (SUBCYCLE EQ 1) {
```

```
                    }
                    IF (SUBCYCLE EQ 2) {
                    }
                    IF (SUBCYCLE EQ 3) {
                    }
                    IF (SUBCYCLE EQ 4) {
                            IF (OBJ EQ 0) MPC = ADDRF;
                    }
                }

                ELSE {

                    IF (SUBCYCLE EQ 1) {

                        SWITCH (PNAME) {
                          CASE   1 : TEMP = A;BREAK;
                          CASE   2 : TEMP = B;BREAK;
                          CASE   4 : TEMP = C;BREAK;
                          CASE   8 : TEMP = D;BREAK;
                          CASE  16 : TEMP = MBR;BREAK;
                          CASE  32 : TEMP = X;BREAK;
                          CASE  64 : TEMP = IR;BREAK;
                          CASE 128 : TEMP = CZERO;BREAK;
                        }
                    }
                IF (SUBCYCLE EQ 2 ) {

                }

                IF (SUBCYCLE EQ 3 ) {
                }
                IF (SUBCYCLE EQ 4) {

                        TEMP = TEMP   BITNUM;
                        IF (((OBJ EQ 1)  (TEMP GT 0))
                                OR
                            ((OBJ EQ 0)  (TEMP EQ 0)))
                        MPC = ADDRF;
                        ELSE
                                MPC = MPC + 1;
                }

            }
        }

}
@@
END
```

This section describes the implementation of the AMISS version of the microprogramming language described by Parker. The

language was duplicated exactly except for the following:

1) The test statement syntax was changed from the form "IF BIT(NUMBER,REGISTER) THEN GOTO LABEL" to "IF BIT(REGISTER,NUMBER) GOTO LABEL;

2) A statement of the form "R=R+(-1)" was changed to "R=R-1";

3) a "," rather than a ";" is used in the microprogram to separate microinstructions if more than one microinstruction is used to generate a control word;

4) a ";" rather than a ";;" is used to separate control words;

5) INV (REGISTER) instead of COMPLEMENT (REGISTER).

The following is the description of the microprogramming language that is submitted to MPL:

```
DEFINE TEST
{
%%
MMTEST:          IF BIT "(" one "," two ")" GOTO LABELOPER
                        {OPCODE = 1;
                         OBJ = 1;
                         ADDRF = MMADDRESS;}
          |      GOTO LABELOPER2
                        {OPCODE = 1;
                         OBJ = 0;
                         RNAME = 128;
                         BITNUM = 1;
                         ADDRF = MMADDRESS;
                        }
          ;

one:      |      A          { RNAME = 1; }
          |      B          { RNAME = 2; }
          |      C          { RNAME = 4; }
          |      D          { RNAME = 8; }
          |      MBR        { RNAME = 16;}
          |      X          { RNAME = 32;}
          |      IR         { RNAME = 64;}
          |      CZERO      { RNAME = 128;}
          ;
two:             "0"        { BITNUM = 1; }
          |      "1"        { BITNUM = 2; }
          |      "2"        { BITNUM = 4; }
          |      "3"        { BITNUM = 8; }
          |      "4"        { BITNUM = 16;}
          |      "5"        { BITNUM = 32;}
          |      "6"        { BITNUM = 64;}
          |      "7"        { BITNUM = 128;}
          |      "8"        { BITNUM = 256;}
          |      "9"        { BITNUM = 512;}
          |      "1" "0"    { BITNUM = 1024;}
          |      "1" "1"    { BITNUM = 2048;}
          |      "1" "2"    { BITNUM = 4096;}
          |      "1" "3"    { BITNUM = 8192;}
          |      "1" "4"    { BITNUM = 16384;}
          |      "1" "5"    { BITNUM = 32768;}
          ;
%%
}

|         RD               {OPCODE = 0;
                          RW = 1;
                          }
|         WR               {OPCODE = 0;
                          RW = 2;
                          }
|         MAR "=" PC       {OPCODE = 0;
                           PCMAR = 1;
                          }
```

```
|       IR "=" MBR          {OPCODE = 0;
                            MBRIR = 1;
                           }
|       PC "=" IR                    {OPCODE = 0;
                            PCIR = 1;
                           }
|       MAR "=" SP         {OPCODE = 0;
                            SPMAR = 1;
                           }
|       MAR "=" IR         {OPCODE = 0;
                            IRMAR = 1;
                           }
|       PC "=" PC "+" "1"           {OPCODE = 0;
                           ACNTRL = 4;
                           BCNTRL = 8;
                           ALUF = 1;
                           PCFEED = 1;
                           }
|       SP "=" SP "+" "1"           {OPCODE = 0;
                           ACNTRL = 2;
                           BCNTRL = 8;
                           ALUF = 1;
                           SPFEED = 1;
                           }
|       SP "=" SP "-" "1"  {OPCODE = 0;
                            ACNTRL = 2;
                            BCNTRL = 4;
                            ALUF = 1;
                            SPFEED = 1;
                           }
|       A "=" INV "("A")" "+" MBR          {OPCODE = 0;
                                ACNTRL = 8;
                                BCNTRL = 16;
                                CMPL = 1;
                                ALUF = 1;
                                AFEED = 1;
                               }
|       MBR "=" A "+" "1"          {OPCODE = 0;
                            ACNTRL = 8;
                            BCNTRL = 8;
                            ALUF = 1;
                            DBUSFEED = 1;
                           }
|       A "=" MBR                  {OPCODE = 0;
                            REG31 = 1;
                           }
|       MBR "=" A "+" MBR          {OPCODE = 0;
                           ACNTRL = 8;
                           BCNTRL = 16;
                           ALUF = 1;
                           DBUSFEED = 1;
                          }
```

Parker presents a microprogram to execute a small stack oriented instruction set. For this simulation, a HALT instruction was added. The instruction set is:

OPCODE INSTRUCTION

000 PUSH the low 13 bits of the instruction

001 POP the top of the stack value into the A register

010 HALT

100 ADD the top two stack values, result on top

101 SUBtract the second from the top value from the top value, result on top

110 STOre the (Top-1) value into the memory location given by the top

111 GET the contents of the address given by the top of the stack, result goes on Top.

The following is a microprogram written in the microprogramming language adapted from the one described by Parker. Aside from the change in the syntax of the microprogramming language and the section for decoding the HALT instruction, it is the same microprogram presented by Parker.

```
START:
FETCH:  MAR = PC,RD;
        IR = MBR,PC = PC + 1;
        IF BIT (IR,15) GOTO AROPS;
        IF BIT (IR,14) GOTO MTEST;
        IF BIT (IR,13) GOTO POP;

MTEST:  IF BIT(IR,13) GOTO FETCH;
        GOTO HALT;

PUSH:   SP = SP + 1;
        MAR = SP, WR;
        GOTO FETCH;

POP:    MAR = SP, SP = SP - 1,RD;
        A = MBR;
        GOTO FETCH;

AROPS:  IF BIT(IR,14) GOTO MEMOP;
        IF BIT(IR,13) GOTO SUB;

ADD:    MAR = SP, SP = SP - 1, RD;
        A = MBR, MAR = SP, RD;
        MAR = SP, MBR = A + MBR, WR;
        GOTO FETCH;

SUB:    MAR = SP, SP = SP - 1, RD;
        A = MBR, MAR = SP, RD;
        A = INV(A) + MBR;
        MBR = A + 1, WR;
        GOTO FETCH;

MEMOP:  IF BIT(IR,13) GOTO GET;

STO:    MAR = SP, SP = SP - 1, RD;
        IR = MBR,MAR = SP,SP = SP - 1, RD;
        MAR = IR, WR;
        GOTO FETCH;

GET:    MAR = SP, RD;
        IR = MBR;
        MAR = IR, RD;

        MAR = SP, WR;
        GOTO FETCH;
HALT:   GOTO ;
```

Parker presents a program written in the stack based instruction set to be executed by the microprogram. The program is:

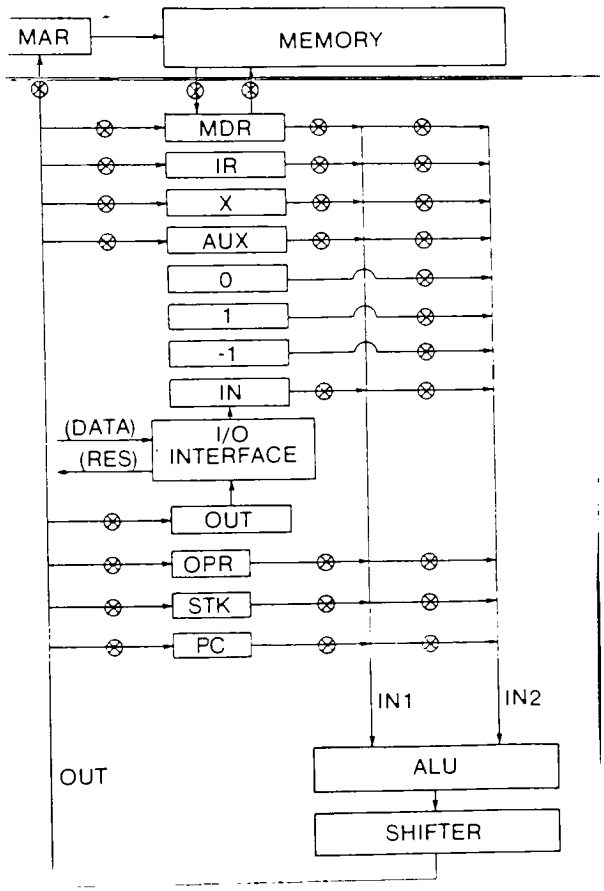
```
0001    PUSH 1
0002    PUSH 2
8000    ADD
0007    PUSH 7
8000    ADD
0005    PUSH 5
e000    SUB
0000    PUSH 0
c000    STO
0000    PUSH 0
e000    GET
0000    PUSH 0
e000    GET
8000    ADD
0000    PUSH 0
c000    STO
0000    PUSH 0
4000    HALT
```

The AMISS hardware using the microprogram interpreter successfully executed this program.

TEST CASE 2

Lubomir Bic presents a microprogrammable computer simulator that was developed at the University of California, Irvine [7]. The architecture consists of a memory unit, an arithmetic/logic unit, a shift unit, an input/output interface, a set of 24 bit registers, and a set of 12 bit registers. Figure 1 is a diagram of the architecture.

Figure 1. Bic's processor.



The architecture is vertically controlled using a 24 bit control word, and two control word formats: a GATE format and a TEST format. AMISS was used to write a simulator for this architecture exactly as described by Bic except for the following:

1) The AMISS OPCODE field was added to each format.

2) A single processor cycle was divided into four subcycles.

3) The input/output interface was simulated by calling two C functions from the hardware description: input() and output(). These 2 functions exist in the files input.c and outputs.c. Provision for including these files is made by naming them in the CFUNCTIONS section of the HDL.

4) An integer variable named temp is used.

The following is a hardware description of the processor:

```
@@ BEGIN
@ CFUNCTIONS
        <VARS  int      temp;                   >
        <MAKEFILES      output.o input.c        >
@ FORMATS
        FORMAT1:        <2> = OPCODE,
                        <4> = MEMOP,
                        <4> = SHIFTOP,
                        <4> = OUT,
                        <4> = IN2,
                        <4> = IN1,
                        <4> = OPC1;

        FORMAT2:        <2> = OPCODE,
                        <12> = ADDRE,
                        <3> = I,
                        <3> = OPC2;


@ PARTS
        <24> =  MBR, IR, X, MAR,
                AUX, CZERO, CONE, CMINUSONE,
                INREG, OUTREG,
                IN1BUS, IN2BUS, OUTBUS,
                ALU, ALUPORT1, ALUPORT2,
                SHIFTER;

        <12> =  OPR, STK, PC;
```

```
        <8>  =  MPC;

@ MEMORY
        <128><29> = CMEM;
        <1024><24> = MEM;

@ INITIALIZATIONS
        CONE = 1;
        CZERO = 0;
        CMINUSONE = 0xffffff;
        STK = 511;

@ MICROENGINE
{

IF ( OPCODE EQ 0 ) {

    IF ( SUBCYCLE EQ 1 ) {

        SELECT1( IN1BUS, IN1, 13:
                IN1BUS, IN1BUS, IN1BUS,
                MBR, IR, X, AUX, IN1BUS,
                IN1BUS, IN1BUS, OPR, STK, PC);

        IF ( IN1 EQ 7 ) {
                INPUT();
                IN1BUS = INREG;
        }

        SELECT2( IN2BUS, IN2, 13:
                CZERO, CONE, CMINUSONE,
                MBR, IR, X, AUX, IN2BUS,
                IN2BUS, IN2BUS,
                OPR, STK, PC);

        IF ( IN2 EQ 7) {
                INPUT();
                IN2BUS = INREG;
        }

        ALUPORT1 = IN1;
        ALUPORT2 = IN2;

    }

    IF ( SUBCYCLE EQ 2 ) {

        SWITCH ( OPC1 ) {

        CASE 0: ALU = ALUPORT1 + ALUPORT2; BREAK;
        CASE 1: ALU = ALUPORT1 - ALUPORT2; BREAK;
        CASE 2: ALU = ALUPORT1  ALUPORT2; BREAK;
        CASE 3: ALU = ALUPORT1 OR ALUPORT2; BREAK;
```

```
            CASE 4: ALU = ALUPORT1 XOR ALUPORT2; BREAK;
            }

            SHIFTER = ALU;
      }

      IF ( SUBCYCLE EQ 3 ) {

            SWITCH ( SHIFTOP ) {
                    CASE 1: SHIFTER SL 1; BREAK;
                    CASE 2: SHIFTER SR 1; BREAK;
            }

            OUTBUS = SHIFTER;
      }

      IF ( SUBCYCLE EQ 4 ) {

            SWITCH ( OUT ) {
                    CASE 3: MBR = OUTBUS; BREAK;
                    CASE 4: IR = OUTBUS; BREAK;
                    CASE 5: X = OUTBUS; BREAK;
                    CASE 6: AUX = OUTBUS; BREAK;
                    CASE 8: OUTREG = OUTBUS;
                            OUTPUT();
                            BREAK;
                    CASE 9: MAR = OUTBUS; BREAK;
                    CASE 10: OPR = OUTBUS; BREAK;
                    CASE 11: STK = OUTBUS; BREAK;
                    CASE 12: PC = OUTBUS; BREAK;
            }

            IF (MEMOP EQ 1) MBR = MEMORY[MAR];
            IF (MEMOP EQ 2) MEMORY[MAR] = MBR;

            MPC = MPC + 1;
      }

      RETURN;
}

IF ( OPCODE EQ 1 ) {

            IF (SUBCYCLE EQ 1)
                    ;
            IF (SUBCYCLE EQ 2)
                    ;
            IF (SUBCYCLE EQ 3)
                    ;
            IF (SUBCYCLE EQ 4) {

               SELECT1(TEMP,I,24:
                0x1, 0x2, 0x4, 0x8,
                0x10,0x20,0x40,0x80,
```

```
        0x100,0x200,0x400,0x800,
        0x1000,0x2000,0x4000,0x8000,
        0x10000,0x20000,0x40000,0x80000
        0x100000,0x200000,0x400000,0x800000);

    SWITCH ( OPC2 ) {
          CASE 5: MPC = ADDRF; BREAK;
          CASE 6:
                  TEMP = MBR  TEMP;
                  IF ( TEMP EQ 0 ) MPC = ADDRF;
                  ELSE
                          MPC = MPC + 1;
                  BREAK;
          CASE 7:
                  TEMP = IR  TEMP;
                  IF ( TEMP EQ 0 ) MPC = ADDRF;
                  ELSE
                          MPC = MPC + 1;
                  BREAK;
          DEFAULT: BREAK;
          }
  }

}

@@
END
```

The I/O interface is simulated by the two files, input.c and output.c:

File input.c:

```
#include        "vars_defines.h"

input()
{
   printf("***** INPUT: n");
   scanf("%x",&inrec);
}
```

File output.c:

```
#include        "variables.h"

output()                {
printf("***** OUTPUT: %xn",outrec);          }
```

A microprogramming language is presented. Figure 2 contains the instruction set of the microprogramming language.

THE INSTRUCTION SET OF THE MICROPROGRAMMING LANGUAGE

Executable Instructions

code	instruction	description
0	ADD	ADD in1 to in2, store result in out
1	SUB	Subtract in1 from in2, store result in out
2	AND	AND in1 with in2, store result in out
3	OR	OR in1 with in2, store result in out
4	EOR	Excl. OR in1 with in2, store result in out
5	BR M	Branch to location M
6	BMDR i,M	Branch to location M if bit i of MDR is zero
7	BRIR i,M	Branch to location M if bit i of IR is zero
8	HALT	Terminate execution

Figure 2. Bic's instruction set.

This language was translated to statements acceptable to the AMISS microprogramming language generator MPL resulting in a language having exactly the same capabilities but with a different syntax.
The new syntax is found in Figure 3:

```
code  instruction
-----------------
0     PART1 "=" PART2 "+" PART3
1     PART1 "=" PART2 "-" PART3
2     PART1 "=" PART2  PART3
3     PART1 "=" PART2 OR PART3
4     PART1 "=" PART2 EOR PART3
5     GOTO LABELOPER
6     IF BIT "(" MDR "," num ")" GOTO LABELOPER
7     IF BIT "(" IR  "," num ")" GOTO LABELOPER
8     HALT
```

The following is the input to MPL that generates the microassembler:

```
DEFINE TEST
{
%%
MMTEST:          I= BIT "(" one "," two ")" GOTO LABELOPER
                        {OPCODE = 1;
                         ADDRF = MMADDRESS;}

        |        GOTO LABELOPER
                        {OPCODE = 1;
                         OPC2 = 5;
                         ADDRF = MMADDRESS;
                        }
        ;

one:             "BR        { OPC2 = 6;}
        |        IR         { OPC2 = 7;}
        ;
two:             "0"        { I   = 0; }
        |        "1"        { I   = 1; }
        |        "2"        { I   = 2; }
        |        "3"        { I   = 3; }
        |        "4"        { I   = 4;}
        |        "5"        { I   = 5;}
        |        "6"        { I   = 6;}
        |        "7"        { I   = 7;}
        |        "8"        { I   = 8;}
        |        "9"        { I   = 9;}
        |        "1" "0"    { I   = 10;}
        |        "1" "1"    { I   = 11;}
        |        "1" "2"    { I   = 12;}
        |        "1" "3"    { I   = 13;}
        |        "1" "4"    { I   = 14;}
        |        "1" "5"    { I   = 15;}
        |        "1" "6"    { I   = 16;}
        |        "1" "7"    { I   = 17;}
        |        "1" "8"    { I   = 18;}
        |        "1" "9"    { I   = 19;}
        |        "2" "0"    { I   = 20;}
        |        "2" "1"    { I   = 21;}
        |        "2" "2"    { I   = 22;}
        |        "2" "3"    { I   = 23;}
        ;

%%
}
|        MBR "=" MBR EOR "-" "1" {OPCODE = 0;
                                  IN1 = 3;
                                  IN2 = 2;
```

```
                                    OPC1 = 4;
                                    OUT = 3;
                                    }
|       MBR "=" MBR AND AUX         {OPCODE = 0;
                                    IN1 = 3;
                                    IN2 = 6;
                                    OPC1 = 2;
                                    OUT = 3;
                                    }
|       MBR "=" MBR OR AUX          {OPCODE = 0;
                                     IN1 = 3;
                                     IN2 = 6;
                                     OPC1 = 3;
                                     OUT = 3;
                                    }
|       MBR "=" MBR XOR AUX         {OPCODE = 0;
                                     IN1 = 3;
                                     IN2 = 6;
                                     OPC1 = 4;
                                     OUT = 3;
                                    }
|       STK "=" STK "+" "1"         {OPCODE = 0;
                                    IN1 = 11;
                                    IN2 = 1;
                                    OPC1 = 0;
                                    OUT = 11;
                                    }


|       RD                  {OPCODE = 0;
                            MEMOP = 1;
                            }
|       WR                  {OPCODE = 0;
                            MEMOP = 2;
                            }
|       LSHIFT              {OPCODE = 0;
                             SHIFTOP = 1;
                            }
|       RSHIFT              {OPCODE = 0;
                            SHIFTOP = 2;
                            }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 3, OPC1 = 0, IN2 = 0,
                                ALUF = 0;
            SELECT(MBR,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 4, OPC1 = 0, IN2 = 0,
                                ALUF = 0;
            SELECT(IR,IN1,13:
                    CZERO,CONE,CMINUSONE,
```

```
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 5, OPC1 = 0, IN2 = 0,
                       ALUF = 0;
            SELECT(X,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 6, OPC1 = 0, IN2 = 0,
                       ALUF = 0;
            SELECT(AUX,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 7, OPC1 = 0, IN2 = 0,
                       ALUF = 0;
            SELECT(MAR,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 10, OPC1 = 0, IN2 = 0,
                       ALUF = 0;
            SELECT(OPR,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 11, OPC1 = 0, IN2 = 0,
                       ALUF = 0;

            SELECT(STK,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MAR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 12, OPC1 = 0, IN2 = 0,
                       ALUF = 0;
            SELECT(PC,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MAR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }
|       DEFINE SEMANTICS {
            CONSTANTS: OPCODE = 0, OUT = 8, OPC1 = 0, IN2 = 0,
```

```
                    ALUF = 0;
                SELECT(OUTREG,IN1,13:
                    CZERO,CONE,CMINUSONE,
                    MBR,IR,X,
                    AUX,INREG,OUTREG,MAR,OPR,STK,PC)
        }

|       PC "=" PC "+" "1"          {OPCODE = 0;
                                   IN1 = 12;
                                   IN2 = 1;
                                   OPC1 = 0;
                                   OUT = 12;
                                   }
|       AUX "=" AUX "-" AUX       {OPCODE = 0;
                                   IN1 = 6;
                                   IN2 = 6;
                                   OPC1 = 1;
                                   OUT = 6;
                                   }
|       OPR "=" OPR "+" X         {OPCODE = 0;
                                   IN1 = 10;
                                   IN2 = 3;
                                   OPC1 = 0;
                                   OUT = 10;
                                   }
|       STK "=" STK "-" "1"       {OPCODE = 0;
                                   IN1 = 11;
                                   IN2 = 2;
                                   OPC1 = 1;
                                   OUT = 11;
                                   }
|       MBR "=" MBR "+" "1"       {OPCODE = 0;
                                   IN1 = 3;
                                   IN2 = 1;
                                   OPC1 = 0;
                                   OUT = 3;
                                   }
|       MBR "=" MBR "-" AUX       {OPCODE = 0;
                                   IN1 = 3;
                                   IN2 = 6;
                                   OPC1 = 1;
                                   OUT = 3;
                                   }
|       MBR "=" MBR "+" AUX {OPCODE = 0;
                                   IN1 = 3;
                                   IN2 = 6;
                                   OPC1 = 0;
                                   OUT = 3;
                                   }
|       MAR "=" STK "-" "1"       {OPCODE = 0;
                                   IN1 = 11;
                                   IN2 = 1;
                                   OPC1 = 0;
                                   OUT = 9;
```

```
                                    }
| X "=" X "+" "1"                   {OPCODE = 0;
                                    IN1 = 5;
                                    IN2 = 1;
                                    OPC1 = 0;
                                    OUT = 5;
                                    }
| X "=" X "-" "1"                   {OPCODE = 0;
                                    IN1 = 5;
                                    IN2 = 1;
                                    OPC1 = 1;
                                    OUT = 5;
                              }
```

Bic presents a microprogram to interpret a stack based instructions set. This instruction set is in Figure 4.

Figure 4. Bic's instruction set. Continued on the following page.

One-Operand Instructions

code	instruction		stack length	description
00	LOAD	M	+1	Load stack from memory location M
01	LDI	c	+1	Load stack with constant c
02	STORE	M	−1	Store tos to memory location M
03	BR	M	0	Branch to location M
04	BPL	M	0	Branch to M if tos is > or = zero
05	BMI	M	0	Branch to M if tos is < zero
06	BZE	M	0	Branch to M if tos is = zero
07	BRSUB	M	+1	Branch to M and save return address on stack
08	BXPL	M	0	Branch to M if X is > or = zero
09	BXMI	M	0	Branch to M if X is < zero
0A	BXZE	M	0	Branch to M if X is = zero
0B	LDX	M	0	Load X from memory location M
0C	LDXI	c	0	Load X with constant c
0D	INPUT	M	0	Input value from file DATA to location M
0E	OUTPUT	M	0	Output value from location M to file RES

Zero-Operand Instructions

code	instruction	stack length	description
20	ADD	−1	Add 2 top-most values of stack
21	SUB	−1	Subtract tos from value next below tos
22	AND	−1	AND 2 top-most values of stack
23	OR	−1	OR 2 top-most values of stack
24	EOR	−1	Excl. OR 2 top-most values of stack
25	COMP	0	Complement tos (one's complement)
26	SHL	0	Shift left tos by one
27	SHR	0	Shift right tos by one
28	TXS	+1	Transfer X to tos
29	TSX	−1	Transfer tos to X
2A	INX	0	Increment X by one
2B	DEX	0	Decrement X by one
2C	RET	−1	Return from subroutine
2D	POP	−1	Discard tos
2E	HALT	0	Terminate execution

Figure 4. Bic's instruction set.

The microprogram that interprets this instruction set is in Figure 5.

Figure 5. Microprogram presented by Bic to interpret the instruction set in Figure 4. Continued on the next page.

```
LN  LOC  OBJECT  SOURCE-PROGRAM
 1  *** INTERPRETER FOR THE STANDARD MICOS

 2  00  0C0801  FETCH   ADD     PC,0,MAR;R
 3  01  030400          ADD     MDR,0,IR
 4  02  0C1C00          ADD     PC,1,PC
 5  03  030A00          ADD     MDR,0,OPR
 6  04  166600          SUB     AUX,AUX,AUX
 7  05  711028          BRIR    17,ONEOPR
 8  06  710014          BRIR    16,I2
 8  07  70F008          BRIR    15,I22
10  08  500007          BR      NEW38
11  08  70E00F  I22     BRIR    14,I221
12  0A  700000          BRIR    13,I2212
13  0B  70C005          BRIR    12,NEW36
14  0C  500006          BR      NEW37
15  0D  70C003  I2212   BRIR    12,NEW34
16  0E  500004          BR      NEW35
17  0F  700012  I221    BRIR    13,I2211
18  10  70C001          BRIR    12,NEW32
18  11  500002          BR      NEW33
20  12  70C0CF  I2211   BRIR    12,NEW30
21  13  500000          BR      NEW31
22  14  70F020  I2      BRIR    15,I21
23  15  70E018          BRIR    14,I212
24  16  700018          BRIR    13,I2122
25  17  70C0C3          BRIR    12,HALT
26  18  5000CE          BR      NEW2F
27  18  70C080  I2122   BRIR    12,RET
28  1A  5000C1          BR      POP
28  1B  70001E  I212    BRIR    13,I2121
30  1C  70C08B          BRIR    12,INX
31  1D  50008B          BR      DEX
32  1E  70C081  I2121   BRIR    12,TXS
33  1F  500085          BR      TSX
34  20  70E026  I21     BRIR    14,I211
35  21  700024          BRIR    13,I2112
36  22  70C0AB          BRIR    12,SHL
37  23  5000AE          BR      SHR
38  24  70C0A2  I2112   BRIR    12,EOR
38  25  5000AB          BR      COMP
40  26  70002B  I211    BRIR    13,I2111
41  27  70C096          BRIR    12,AND
42  28  50008C          BR      OR
43  28  70C0BA  I2111   BRIR    12,ADD
44  2A  500090          BR      SUB
45  2B  714020  ONEOPR  BRIR    20,INDIR
46  2C  0A5A00          ADD     OPR,X,OPR
47  20  715031  INDIR   BRIR    21,IXDONE
48  2E  0A0901          ADD     OPR,0,MAR;R
48  2F  030A00          ADD     MDR,0,OPR
50  30  500020          BR      INDIR
51  31  71003F  IXDONE  BRIR    16,I1
52  32  70F034          BRIR    15,I12
53  33  5000C0          BR      NEW18
54  34  70E03A  I12     BRIR    14,I121
55  35  700038          BRIR    13,I1212
56  36  70C0C8          BRIR    12,NEW16
57  37  5000CC          BR      NEW17
58  38  70C0C8  I1212   BRIR    12,NEW14
58  3B  5000CA          BR      NEW15
60  3A  700030  I121    BRIR    13,I1211
61  3B  70C0C7          BRIR    12,NEW12
62  3C  5000C8          BR      NEW13
63  3D  70C0C7  I1211   BRIR    12,NEW12
64  3E  5000C6          BR      NEW11
65  3F  70F048  I1      BRIR    15,I11
66  40  70E046          BRIR    14,I112
67  41  700044          BRIR    13,I1122
68  42  70C087          BRIR    12,OUTPUT
68  43  5000C4          BR      NEW0F
70  44  70C082  I1122   BRIR    12,LDXI
71  45  500084          BR      INPUT
72  46  700048  I112    BRIR    13,I1121
73  47  70C07C          BRIR    12,BXZE
74  48  50007F          BR      LDX
75  48  70C076  I1121   BRIR    12,BXPL
76  4A  50007B          BR      BXMI
77  4B  70E051  I11     BRIR    14,I111
78  4C  70004F          BRIR    13,I1112
78  4D  70C06A          BRIR    12,BZE
80  4E  500071          BR      BRSUB
81  4F  70C064  I1112   BRIR    12,BPL
82  50  500067          BR      BMI
83  51  700054  I111    BRIR    13,I1111
84  52  70C05E          BRIR    12,STORE
85  53  500062          BR      BR
86  54  70C056  I1111   BRIR    12,LOAD
87  55  50005A          BR      LDI
88  56  0A0801  LOAD    ADD     OPR,0,MAR;R
88  57  1B1B00          SUB     STK,1,STK
80  58  080802          ADD     STK,0,MAR;W
81  58  500000          BR      FETCH
82  5A  0A0300  LDI     ADD     OPR,0,MDR
83  5B  1B1B00          SUB     STK,1,STK
84  5C  080802          ADD     STK,0,MAR;W
85  50  500000          BR      FETCH
86  5E  080801  STORE   ADD     STK,0,MAR;R
87  5F  0A0802          ADD     OPR,0,MAR;W
88  60  0B1B00          ADD     STK,1,STK
88  61  500000          BR      FETCH
100 62  0A0C00  BR      ADD     OPR,0,PC
101 63  500000          BR      FETCH
102 64  080801  BPL     ADD     STK,0,MAR;R
103 65  617052          BMDR    23,BR
104 66  500000          BR      FETCH
105 67  080801  BMI     ADD     STK,0,MAR;R
106 68  617000          BMDR    23,FETCH
107 68  500062          BR      BR
108 6A  080801  BZE     ADD     STK,0,MAR;R
108 6B  617060          BMDR    23,GRZERO
110 6C  500000          BR      FETCH
```

Figure 5. continued

```
111  6D  432300  GRZERO  EOR   MOR,-1,MOR
112  6E  031300          ADD   MDR,1,MOR
113  6F  617062          BMOR  23,BR
114  70  500000          BR    FETCH
115  71  0C0300  BRSUB   ADD   PC,0,MOR
116  72  181800          SUB   STK,1,STK
117  73  080802          ADD   STK,0,MAR;W
116  74  0A0C00          ADD   OPR,0,PC
119  75  500000          BR    FETCH
120  76  050300  BXPL    ADD   X,0,MOR
121  77  617062          BMOR  23,BR
122  78  500000          BR    FETCH
123  79  050300  BXMI    ADD   X,0,MOR
124  7A  617000          BMOR  23,FETCH
125  7B  500062          BR    BR
126  7C  050300  BXZE    ADD   X,0,MOR
127  7D  617060          BMOR  23,GRZERO
128  7E  500000          BR    FETCH
129  7F  0A0901  LDX     ADD   OPR,0,MAR;R
130  80  030500          ADD   MDR,0,X
131  81  500000          BR    FETCH
132  82  0A0500  LOXI    ADD   OPR,0,X
133  83  500000          BR    FETCH
134  84  070300  INPUT   ADD   IN,0,MOR
135  85  0A0902          ADD   OPR,0,MAR;W
136  66  500000          BR    FETCH
137  87  0A0901  OUTPUT  ADD   OPR,0,MAR;R
138  88  030600          ADD   MOR,0,OUT
139  69  500000          BR    FETCH
140  8A  060901  ADD     ADD   STK,0,HAR;R
141  8B  030600          ADD   MOR,0,AUX
142  8C  081800          ADD   STK,1,STK
143  8D  080801          ADD   STK,0,MAR;R
144  6E  036302          ADD   MOR,AUX,MOR;W
145  6F  500000          BR    FETCH
146  90  080901  SUB     ADD   STK,0,MAR;R
147  91  030600          ADD   MOR,0,AUX
148  92  081800          ADD   STK,1,STK
149  93  080901          ADD   STK,0,MAR;R
150  94  136302          SUB   MOR,AUX,MOR;W
151  95  500000          BR    FETCH
152  96  060901  AND     ADD   STK,0,MAR;R
153  97  030600          ADD   MDR,0,AUX
154  98  081800          ADD   STK,1,STK
155  99  080901          ADD   STK,0,MAR;R
156  9A  236302          AND   MOR,AUX,MOR;W
157  9B  500000          BR    FETCH
158  9C  060901  OR      ADD   STK,0,MAR;R
159  9D  030600          ADD   MOR,0,AUX
160  9E  081800          ADD   STK,1,STK
161  9F  080901          ADD   STK,0,MAR;R
162  A0  336302          OR    MOR,AUX,MOR;W
163  A1  500000          BR    FETCH
164  A2  080901  EOR     ADD   STK,0,MAR;R
165  A3  030600          ADD   MOR,0,AUX
166  A4  081800          ADD   STK,1,STK
167  A5  060901          ADD   STK,0,MAR;R
168  A6  436302          EOR   MOR,AUX,MOR;W
169  A7  500000          BR    FETCH
170  A8  060901  COMP    ADD   STK,0,MAR;R
171  A9  432302          EOR   MOR,-1,MDR;W
172  AA  500000          BR    FETCH
173  AB  080901  SHL     ADD   STK,0,MAR;R
174  AC  030312          ADD   MOR,0,MOR;SHL
175  AD  500000          BR    FETCH
176  AE  080901  SHR     ADD   STK,0,MAR;R
177  AF  030322          ADD   MOR,0,MOR;SHR;W
178  B0  500000          BR    FETCH
179  B1  181900  TXS     SUB   STK,1,MAR
180  B2  050302          ADD   X,0,MOR;W
181  B3  181800          SUB   STK,1,STK
182  B4  500000          BR    FETCH
183  B5  080901  TSX     ADD   STK,0,MAR;R
184  B6  030500          ADD   MOR,0,X
185  B7  081800          ADD   STK,1,STK
186  B8  500000          BR    FETCH
187  B9  051500  INX     ADD   X,1,X
188  BA  500000          BR    FETCH
189  BB  151500  DEX     SUB   X,1,X
190  BC  500000          BR    FETCH
191  BD  080901  RET     ADD   STK,0,MAR;R
192  BE  081800          ADD   STK,1,STK
193  BF  030C00          ADD   MOR,0,PC
194  C0  500000          BR    FETCH
195  C1  081800  POP     ADD   STK,1,STK
196  C2  500000          BR    FETCH
197  C3  800000  HALT    HALT
```

Using the MPL microassembler a microprogram was written that duplicates the microprogram presented by Bic. Figure 6 is the MPL microprogram.

```
START:
FETCH:  MAR = PC, RD;
        IR = MBR;
        PC = PC + 1;
        OPR = MAR;
        AUX = AUX - AUX;
        IF BIT(IR,17) GOTO ONEOPR;
        IF BIT(IR,16) GOTO I2;
        IF BIT(IR,15) GOTO I22;
        GOTO NEW38;
I22:    IF BIT(IR,14) GOTO I221;
        IF BIT(IR,13) GOTO I2212;
        IF BIT(IR,12) GOTO NEW36;
        GOTO NEW37;
I2212:  IF BIT(IR,12) GOTO NEW34;
        GOTO NEW35;
I221:   IF BIT(IR,13) GOTO I2211;
        IF BIT(IR,12) GOTO NEW32;
        GOTO NEW33;
I2211:  IF BIT(IR,12) GOTO NEW30;
        GOTO NEW31;
I2:     IF BIT(IR,15) GOTO I21;
        IF BIT(IR,14) GOTO I212;
        IF BIT(IR,13) GOTO I2122;
        IF BIT(IR,12) GOTO HALT;
        GOTO NEW2F;
I2122:  IF BIT(IR,12) GOTO RET;
        GOTO POP;
I212:   IF BIT(IR,13) GOTO I2121;
        IF BIT(IR,12) GOTO INX;
        GOTO DEX;
I2121:  IF BIT(IR,12) GOTO TXS;
        GOTO TSX;
I21:    IF BIT(IR,14) GOTO I211;
        IF BIT(IR,13) GOTO I2112;
        IF BIT(IR,12) GOTO SHL;
        GOTO SHR;
I2112:  IF BIT(IR,12) GOTO EXOR;
        GOTO COMP;
I211:   IF BIT(IR,13) GOTO I2111;
        IF BIT(IR,12) GOTO A;
        GOTO OOR;
I2111:  IF BIT(IR,12) GOTO ADDD;
        GOTO SUB;
ONEOPR: IF BIT(IR,20) GOTO IR;
        OPR = OPR + X;


IR:     IF BIT(IR,21) GOTO IXDONE;
```

```
          MAR = OPR , RD;
          OPR = MBR ;
          GOTO IR;
IXDONE:   IF BIT(IR,16) GOTO I1;
          IF BIT(IR,15) GOTO I2;
          GOTO NEW13;
I12:      IF BIT(IR,14) GOTO I121;
          IF BIT(IR,13) GOTO I1212;
          IF BIT(IR,12) GOTO NEW16;
          GOTO NEW17;
I1212:    IF BIT(IR,12) GOTO NEW14;
          GOTO NEW15;
I121:     IF BIT(IR,13) GOTO I1211;
          IF BIT(IR,12) GOTO NEW12;
          GOTO NEW13;
I1211:    IF BIT(IR,12) GOTO NEW12;
          GOTO NEW11;
I1:       IF BIT(IR,15) GOTO I11;
          IF BIT(IR,14) GOTO I112;
          IF BIT(IR,13) GOTO I1122;
          IF BIT(IR,12) GOTO OUTPUT;
          GOTO NEW0F;
I1122:    IF BIT(IR,12) GOTO LDXI;
          GOTO INPUT;
I112:     IF BIT(IR,13) GOTO I1121;
          IF BIT(IR,12) GOTO BXZE;
          GOTO LDX;
I1121:    IF BIT(IR,12) GOTO BXPL;
          GOTO BXMI;
I11:      IF BIT(IR,14) GOTO I111;
          IF BIT(IR,13) GOTO I1112;
          IF BIT(IR,12) GOTO BZE;
          GOTO BRSUB;
I1112:    IF BIT(IR,12) GOTO BPL;
          GOTO BMI;
I111:     IF BIT(IR,13) GOTO I1111;
          IF BIT(IR,12) GOTO STORE;
          GOTO BRANCH;
I1111:    IF BIT(IR,12) GOTO LOAD;
          GOTO LDI;
LOAD:     MAR = OPR, RD;
          STK = STK - 1;
          MAR = STK, WR;
          GOTO FETCH;
LDI:      MBR = OPR;
          STK = STK - 1;
          MAR = STK, WR;
          GOTO FETCH;
STORE:    MAR = STK, RD;
          MAR = OPR, WR;
          STK = STK + 1;
          GOTO FETCH;
BRANCH:   PC = OPR;
          GOTO FETCH;
```

```
BPL:     MAR = STK, RD;
         IF BIT(MBR,23) GOTO BRANCH;
         GOTO FETCH;
BMI:     MAR = STK, RD;
         IF BIT(MBR,23) GOTO FETCH;
         GOTO BRANCH;
BZE:     MAR = STK, RD;
         IF BIT(MBR,23) GOTO BRZERO;
         GOTO FETCH;

BRZERO:  MBR = MBR EOR -1;
         MBR = MBR + 1;
         IF BIT(MBR,23) GOTO BRANCH;
         GOTO FETCH;
BRSUB:   MBR = PC;
         STK = STK - 1;
         MAR = STK, WR;
         PC = OPR;
         GOTO FETCH;
BXPL:    MBR = X;
         IF BIT(MBR,23) GOTO BRANCH;
         GOTO FETCH;
BXMI:    MBR = X;
         IF BIT(MBR,23) GOTO FETCH;
         GOTO BRANCH;
BXZE:    MBR = X;
         IF BIT(MBR,23) GOTO BRZERO;
         GOTO FETCH;
LDX:      MAR = OPR, RD;
         X = MBR;
         GOTO FETCH;
LDXI:    X = OPR;
         GOTO FETCH;
INPUT:   MBR = INREG;
         MAR = OPR, WR;
         GOTO FETCH;
OUTPUT:  MAR = OPR, RD;
         OUTREG = MBR;
         GOTO FETCH;
ADDD:    MAR = STK, RD;
         AUX = MBR;
         STK = STK + 1;
         MAR = STK, RD;
         MBR = MBR + AUX, WR;
         GOTO FETCH;
SUB:     MAR = STK, RD;
         AUX = MBR;
         STK = STK + 1;
         MAR = STK, RD;
         MBR = MBR - AUX, WR;
         GOTO FETCH;
A:       MAR = STK, RD;
         AUX = MBR;
         STK = STK + 1;
```

```
            MAR = STK, RD;
            MBR = MBR + AUX, WR;
            GOTO FETCH;
OOR:        MAR = STK, RD;
            AUX = MBR;
            STK = STK + 1;
            MAR = STK, RD;
            MBR = MBR OR AUX, WR;
            GOTO FETCH;
EXOR:       MAR = STK, RD;
            AUX = MBR;
            STK = STK + 1;
            MAR = STK, RD;
            MBR = MBR EOR AUX, WR;
            GOTO FETCH;
COMP:       MAR = STK, RD;
            MBR = MBR EOR -1, WR;
            GOTO FETCH;
SHL:        MAR = STK, RD;
            MBR = MBR, LSHIFT, WR;
            GOTO FETCH;
SHR:        MAR = STK, RD;
            MBR = MBR, RSHIFT, WR;
            GOTO FETCH;
TXS:        MAR = STK - 1;
            MBR = X, WR;
            STK = STK - 1;
            GOTO FETCH;
TSX:        MAR = STK, RD;
            X = MBR;
            STK = STK + 1;
            GOTO FETCH;
INX:        X = X + 1;
            GOTO FETCH;
DEX:        X = X - 1;
            GOTO FETCH;
RET:        MAR = STK, RD;
            STK = STK + 1;
            PC = MBR;
            GOTO FETCH;
POP:        STK = STK + 1;
            GOTO FETCH;
HALT:       GOTO END;

NEW11:      GOTO END;
NEW12:      GOTO END;
NEW13:      GOTO END;
NEW14:      GOTO END;
NEW15:      GOTO END;
NEW16:      GOTO END;
NEW17:      GOTO END;
NEW18:      GOTO END;
NEW0F:      GOTO END;
NEW2F:      GOTO END;
```

```
NEW30:   GOTO END;
NEW31:   GOTO END;
NEW32:   GOTO END;
NEW33:   GOTO END;
NEW34:   GOTO END;
NEW35:   GOTO END;
NEW36:   GOTO END;
NEW37:   GOTO END;
NEW38:   GOTO END;
END:
```

The control memory resulting from the input of this microprogram was used with the simulator to successfully execute programs written using the stack based instruction set. Figure 5 is a sample program that finds the maximum of 5 integers.

```
LN  LOC  OBJECT  SOURCE-PROGRAM
-------------------------------
 1                * PROGRAM TO FIND THE MAXIMUM OF 5 INTEGERS
 2  000  00C004          LDXI   4          **PRESET X TO 5 ITERAT.
 3  001  10D012  NEXTIN  INPUT  A,X        **READ 5 NUMBERS
 4  002  02B000          DEX                *INTO ARRAY A
 5  003  008001          BXPL   NEXTIN
 6  004  00C004          LDXI   4          **PRESET X TO 5 ITERAT.
 7  005  100012          LOAD   A,X        **MAKE LAST EL. OF A TO
 8  006  002017          STORE  MAX         *CURRENT MAX
 9  007  02B000  NEXTNO  DEX               **IF ALL ELEMENTS
10  008  009010          BXMI   OUTPUT      *PROCESSED-OUTPUT MAX
11  009  000017          LOAD   MAX        **COMPARE MAX TO
12  00A  100012          LOAD   A,X         *CURR. EL. OF A
13  00B  021000          SUB                *
14  00C  004007          BPL    NEXTNO      *IF MAX GREATER-CONT.
15  00D  100012          LOAD   A,X         *ELSE OVERWRITE MAX
16  00E  002017          STORE  MAX         *WITH CURR.EL. OF A
17  00F  003007          BR     NEXTNO     **REPEATE FOR NEXT EL.
18  010  00E017  OUTPUT  OUTPUT MAX
19  011  02E000          HALT
20  012  000000  A       BLOCK  5          **ARRAY A
21  017  000000  MAX     BLOCK  1          **CURRENT MAXIMUM

*** ASSEMBLY COMPLETED ***
    0  ERRORS DETECTED
```

Figure 6. Sample assembly language program run by the simulator.

APPENDIX 1: RESERVED WORDS

The following lists contain the reserved words of HDL and MPL. It is suggested that reserved MPL words not be redefined in an HDL program and reserved HDL words not be redefined in an MPL program.

A. HDL RESERVED WORDS.

These words should not be redefined in an HDL program.

BEGIN	END	INITIALIZATIONS
NUM	IF	ELSE
SWITCH	CASE	DEFAULT
BREAK	RETURN	EQ
NE	GT	GE
LT	LE	AND
OR	EOR	NOT
SL	SR	DIV
ID	SELECT1	SELECT2
SELECT3	SELECT4	DEFINE
CONSTANTS	MICROENGINE	FORMATS
FORMAT1	FORMAT2	FORMAT3
MEM	CMEM	CFUNCTIONS
VARS	INCLUDE	MAKEFILES
DEFINES	OPCODE	BAND BOR

B. MPL RESERVED WORDS

The following list contains the reserved words of MPL. These words should not be redefined in an MPL program.

GOTO	SEMANTICS	SELECT
TEST	LSHIFT	RSHIFT
INV	BAND	RO
WR	CONT	CONTFORMAT
THEN	OPCODE	BAND

APPENDIX 2. SYNTAX OF HOL and MPL.

MPL SYNTAX

```
program::=      list

list::=
        |       list @ @ BEGIN @  define_Cfunct define_formats
                        define_parts define_memory
                         def_inits define_rts @ @  E

define_formats::=  FORMATS format_descr @

format_descr::=    formatnum : < NUM > = OPCODE ,  format_specs  ;
        |   format_descr formatnum : < NUM > = OPCODE ,   format_specs ;

formatnum::=    FORMAT1 | FORMAT2 | FORMAT3 | FORMAT4

format_specs::= < NUM > = expr
        |       format_specs , < NUM > = expr


def_inits::=
        |       INITIALIZATIONS
                        init_list  @

init_list::=    id = NUM ;
        |       id [ NUM ] = NUM ;
        |       init_list id = NUM ;
        |       init_list id [ NUM ] = NUM ;

define_memory::=  MEMORY mem_specs @

mem_specs::=    < NUM > < NUM > = memname ;
                        < NUM > < NUM > =  memname ;

memname::=      MEM | CMEM

define_Cfunct::=
        |       CFUNCTIONS
                        Clist @

Clist::=
        |       makefiles
        |       defines
        |       defines makefiles
        |       includes
        |       includesmakefiles
        |       includes defines
        |       includes defines makefiles
        |       vars
        |       vars makefiles
        |       vars defines
        |       vars defines makefiles
```

```
        |       vars includes
        |       vars includes makefiles
        |       vars includes defines
        |       vars includes defines makefiles


vars::=         < VARS  >

includes::=     < INCLUDES  >

defines::=      < DEFINES  >

makefiles::=    < MAKEFILES >

define_parts::= PARTS    parts_list 0

parts_list::=   < NUM > - name_list ;
        |       parts_list < NUM > = name_list ;

name_list::=    expr
        |       name_list , expr

define_rts::=   MICROENGINE     {       rts_list        }

rts_list::=     st
        |       label st
        |       rts_list st
        |       rts_list label st

label::=        id :

st::=           compound_st
        |       { }
        |       expr ;
        |       IF ( expr ) st st
        |       SWITCH ( expr ) st
        |       CASE const_expr : st
        |       DEFAULT : st
        |       BREAK ;
        |       RETURN ;
        |       select ( selectlist ) ;
        |       ;


select::=       SELECT1
        |       SELECT2
        |       SELECT3
        |       SELECT4

selectlist::-   primary , primary , idlist

idlist::-       NUM : primary
        |       idlist , primary
```

```
s::=
          |         ELSE st

compound_st::=    { st_list }

st_list::=        st
          |         st st_list

expr::=
          |         primary
          |         expr binop expr
          |         expr = expr
          |         lvalue = expr
          |         unop expr

primary::=        lvalue
          |         const_expr
          |         ( expr )
          |         primary [ expr ]
          |         primary ( expr )

lvalue::=         id

const_expr::=     NUM

binop::=          EQ
          |         GT
          |         GE
          |         LT
          |         LE
          |         NE
          |         AND
          |         OR
          |         BAND
          |         BOR
          |         EOR
          |         &
          |         *
          |         +
          |         -
          |         DIV
          |         SL
          |         SR


unop::=           NOT

id::=             ID
          |         MEM
          |         CMEM
          |         OPCODE
```

SYNTAX of MPL.

```
program::=      list

list::=
        |       list define_test rt

define_test::=  DEFINE TEST {
                        }


  rt::=             | pname " = " pname semantics
        |           | pname " = " pname op num semantics
        |           | pname " = " pname op " - " num semantics
        |           | pname " = " pname op pname semantics
        |           | RD semantics
        |           | WR semantics
        |           | LSHIFT semantics
        |           | RSHIFT semantics
        |           | pname " = " LSHIFT " ( " pname " ) "
                                                        semantics
        |           | pname " = " LSHIFT " ( " pname op pname " ) "
                        semantics
        |           | pname " = " RSHIFT " ( " pname " ) "
                        semantics
        |           | pname " = " RSHIFT " ( " pname op pname " ) "
                        semantics
        |           | pname " = " BAND  " ( " pname " , "
                        pname " ) " semantics
        |           | pname " = " BOR " ( " pname " ,"
                        pname " ) " semantics
        |           | pname " = " INV " ( " pname " ) "
                        semantics
        |           | pname " = " INV " ( " pname " ) "
                        op pname semantics
        |           | IF pname  GOTO num  semantics
        |           | IF pname  GOTO id semantics
        |           | GOTO num  semantics
        |           | GOTO id semantics
        |           | DEFINE SEMANTICS {
                              const   SELECT
                              ( pname , pname , num :
                              plist )
                    }
        |           rt | RD semantics
        |           rt | WR semantics
        |           rt | LSHIFT semantics
        |           rt | RSHIFT semantics
        |           rt | pname " = " LSHIFT " ( " pname " ) "
                              semantics
        |           rt | pname " = " LSHIFT " ( " pname op pname
                            " ) "  semantics
```

```
        |       rt | pname " = " LSHIFT " ( " pname " ) "
                        semantics
        |       rt | pname " = " RSHIFT " ( " pname op pname
                    " ) " semantics
        |       rt | pname " = " BAND " ( " pname " , "
                    pname " ) " semantics
        |       rt | pname " = " BOR " ( " pname " , "
                    pname " ) " semantics
        |       rt | pname " = " INV " ( " pname " ) "
                        semantics
        |       rt | pname " = " INV " ( " pname " ) "
                        op pname semantics
        |       rt | pname " = " pname  semantics
        |       rt | pname " = " pname op num  semantics
        |       rt | pname " = " pname op " - "  num semantics
        |       rt | pname " = " pname op pname semantics
        |       rt | IF pname  GOTO num semantics
        |       rt | IF pname  GOTO id semantics
        |       rt | GOTO num  semantics
        |       rt | GOTO id semantics
        |       rt | DEFINE SEMANTICS {
                        const   SELECT
                        ( pname , pname , num :
                        plist )
                    }

num::=          NUM
        |       num NUM
        |       " NUM "
        |       num " NUM "

const::=        CONSTANTS : OPCODE  =  num ;
        |       CONSTANTS : OPCODE = num , clist ;

clist:          pname = num
        |       clist , pname = num

plist:          pname
        |       plist , pname

pname:          id

op:             " + "  | " - " | AND | OR | EOR |
                | BAND | BOR

semantics:
        |       { OPCODE = num ; descr }

descr:          id = num ;
        |       id [ num ] = num ;
        |       CONT ; CONT = num ;
        |       descr  id = num ;
        |       descr  id [ num ] = num ;
```

```
        |       descr  CONT ; CONTFORMAT = num ;

id:             ID
```

APPENDIX 3: AN EXAMPLE PROCESSOR.

This section contains an example microarchitecture that is specified and designed using AMISS.

1. SPECIFICATIONS OF SAMPLE ARCHITECTURE.

For the purpose of example, we will design a small processor with a minimum of elements and a vertically encoded control word. This processor will be a smaller version of the one described by Tannenbaum [6]. It will be a 16 bit machine having 8 registers designated AC, R1, R2, PC, AMASK, CONE, IR, TIR. Each of these will be able to feed two internal busses that we will call the ABUS and the BBUS. The two busses will feed two latches that will capture the data from the busses, the ALATCH and the BLATCH. We will call the arithmetic unit the ALU and assign two input ports to it, ALUPORT1 and ALUPORT2. While the BLATCH feeds ALUPORT2 directly, the ALATCH will feed a multiplexor that we will call the AMUX. There will be two registers that function in main memory accesses, the MAR and the MBR. The other input to the AMUX will be the MBR. The AMUX will select whether the ALUPORT1 is fed from the ABUS or from the MBR. We will allow the MAR to be fed directly from the BBUS. In the microengine description, functions will be assigned to the ALU. For now, let us just assume that the ALU has the ability to act on its input and to derive an output. Let us assign the ability to keep track of the status of the result of an ALU function in some status bits that we will call NBIT, ZBIT, CBIT, and VBIT. We will assign a SHIFTER to our architecture that is fed by the the ALU. The SHIFTER will feed a CBUS which is connected to each of the original registers. We will also allow the SHIFTER to feed the MBR. Finally, AMISS requires that the designer explicitly name the microprogram counter as part of his architecture and that it be named MPC. So our last part is the MPC.

The AMISS specification for the elements of a microarchitecture requires that widths be assigned to all elements. For example, to specify 3 registers that are 16 bits wide

named A,B,C and I that are 8 bits wide named X,Y,Z, the following would be required:

```
<16> = A, B, C;
<8> = X, Y, Z;
```

Note carefully the syntax of the parts specification.

The AMISS specification for our processor is:

```
<16> = AC, R1, R2, PC, AMASK, CONE, IR, TIR,
       ALUPORT1, ALUPORT2, ALU,
       MAR, MBR, AMUX,
       SHIFTER,
       ABUS, BBUS, CBUS;
<8> =  MPC;
<1> =  NBIT, ZBIT, CBIT, VBIT;
```

IMPORTANT: We could list as many parts as we want (up to 99) and name them anything we want BUT one of them must be the MPC.

It will be helpful for later debugging purposes to draw a black box diagram containing the elements of our processor. This diagram is in Figure 1.1.

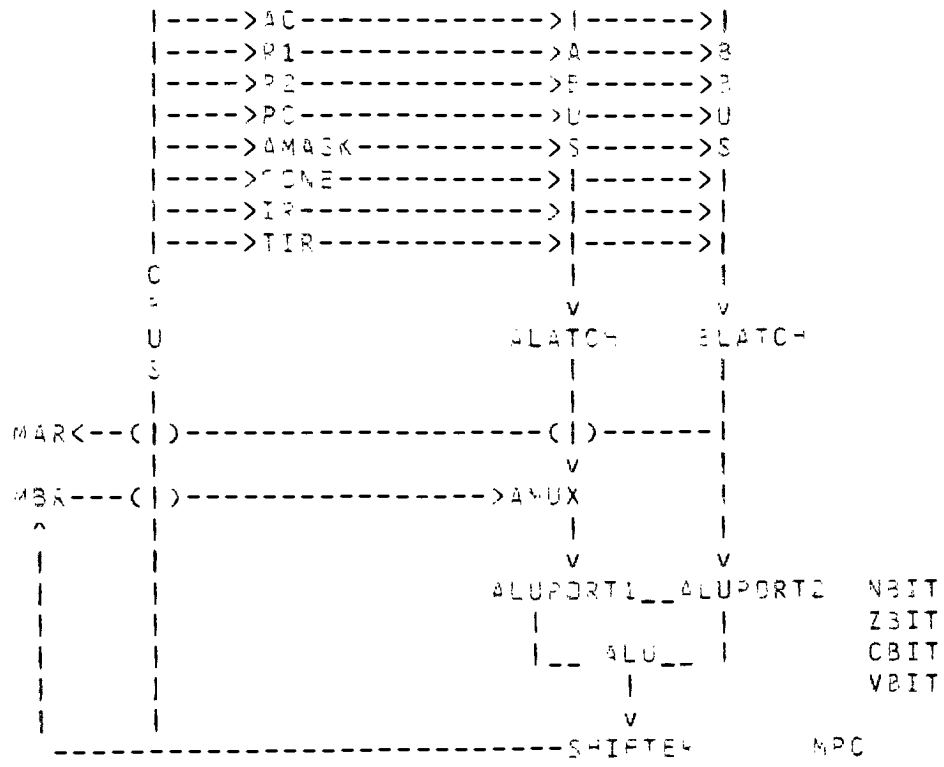


FIGURE 1.1
Block diagram of example processor. No parts for control logic, eg. decoders, are included. The hardware program will become the control logic.

Now that we have described the processor, let us consider a format for the control memory that will control the opening and closing of the gates in the processor. This requires that we first determine how we wish our register transfers to occur. While we do not have to write our hardware descriptor at this point, we must consider how we would like to control the data paths in our processor.

We will call the model upon which to base our control word its format, and since our processor is to be a vertically controlled one, we will attempt to define our control word using just one control word format. This process involves dividing our format into however many fields it takes to control our register transfers. In the AMISS system, this is done by assigning names and widths to the fields. The AMISS system requires that the first field in any format be named the OPCODE. Since all field names must be unique, the name OPCODE cannot be used again within one format description. Let us now consult the diagram of our processor. (FIG. 1.1) Note that we have 8 registers, any of which may feed either the ABUS or the BBUS. In that 3 bits can encode 8 different things, we will assign one field in the control word to control the feeding of the ABUS and one field to the BBUS, each 3 bits wide. Let us name the fields ACNTRL and BCNTRL. The next control point to consider is the AMUX. Since the AMUX has 2 inputs, we can control it using 1 bit. We will call this field the AMUXCNTRL. Likewise, only 1 bit is required to control the feeding of the MAR from the BBUS, the MARCNTRL. Let us say that our ALU has the ability to perform four functions. (We will define them later.) These can be encoded in 2 bits by a field that we will call ALUF. Our next control point is the SHIFTER. We will allow 2 bit in the control word to contol 4 possible shifter functions and call the field SHCNTRL. Next we will assign the ability to control the feeding of the 8 registers of our processor from the CBUS to a 3 bit field called CCNTRL. Also, we will include an enable field that will control whether the gates between the CBUS and the registers will be allowed to be opened. This requires 1 bit and will be encoded in the field called ENC. Since our

processor will want to be able to access main memory, we will designate two 1 bit fields in our control word for this purpose, the READ field and the WRITE field. (This also could be done with one field two bits wide.) Next we will assign a 1 bit field called the MBRCNTRL that will control the feeding of the MBR from the output of the SHIFTER. The last fields to consider are those that serve in the process of deciding how the address of the next control word to be fetched from control memory will be determined. We will need a field in the control word that contains the address to be jumped to in the case that some condition is met. We will also need a field to encode the possible conditions. We will call the former the ADDRF and the latter the COND field. The width of the ADDRF will be determined by the maximum size of our control memory. Let us now decide that that size is 256 words. Thus our ADDRF will have to be 8 bits wide to hold 256 possible addresses. Finally, we will allow the testing of 4 possible conditions so COND will need 2 bits. The AMISS grammar to describe our control word format requires that we give the format a name that consists of the keyword FORMAT followed by number and a colon, thus "FORMAT1:". (If there were a second format it would be "FORMAT2:", and so on up to a maximum of 4 formats.) Also, we must assign the width of the fields and associate them with the field name as, for example, "< 5 > = FIELDNAME". Fieldnames are separated by commas, and the end of a single format description is signaled by a semicolon. So our field description is:

```
FORMAT1:        <2> = OPCODE,
                <8> = ADDRF,
                <3> = ACNTRL,
                <3> = BCNTRL,
                <3> = CCNTRL,
                <1> = END,
                <1> = WRITE,
                <1> = READ,
                <2> = ALUF,
                <1> = AMUXCNTRL,
                <1> = MARCNTRL,
                <1> = MBRCNTRL,
                <2> = SHCNTRL,
                <2> = COND;
```

Next we will assign dimensions to main memory and control memory. We have already determined that our control memory is to be 256 words long and that each word needs 31 bits (the sum of the field widths). Let us say that our main memory is 1024 words along and 16 bits wide. Using the AMISS keywords CMEM and MEM the memory specification is:

```
<256><31> = CMEM;
<1024><16> = MEM;
```

AMISS initially sets all elements of the microarchitecture to hold values of zero. It is possible to initialize elements to whatever values we wish. This would be desirable if there were elements that were to hold constant values for testing purposes or for incrementing. Since the interface to AMISS is in hexidecimal notation, any initializations must be in hex. A hex constant in C is written 0xhhhh... , where h is a hex digit. For example, to set part A to decimal 15 we would say " A = 0xf;". If we had a register named CMINUSONE that was to hold a -1, we would set "CMINUSONE = 0xffff;", the 16 bit, 2's complement hexidecimal represetation of -1. Note that the hex digit 'f' is lower case. Hex numbers are input as lower case, whereas all other input to AMISS except use of the CFUNCTIONS utility must be uppercase. In our example, we will specify that register CONE is 1. Also, we will set the AMASK register to fff. (We will discuss why later.) The AMISS specifiation for this is

```
CONE = 1;
AMASK = 0xfff;
```

Initializations are optional.

The last thing that we will include in our specification is the declaration of a few variables. The reason for doing this is that it is often necessary in describing the register transfers (the actual hardware simulator) to save the results of an assignment statement temporarily so that it can be later tested. We might not want to declare a specific part to save the results so as to avoid "cluttering up" the hardware. While this might decrease the authenticity of the

simulator, it can make it easier to use and to debug. For this purpose, we will include two integer variables named temp and temp2 . These are declared as C varibles with the following syntax:

```
< VARS  int     temp,temp2; >
```

Declaring variables is optional. Note that the AMISS grammar elements "< VARS" and ">" surround the C programming language specification for declaring two integer variables: int temp, temp2;. (Another option here would be to declare the names of any other files that we might wish to to include in the final simulator. Suppose, for example, we wished to have an input interface to our simulator. We could write a file called input.c that contained a C routine that prompted for input. We would list the name of the file as a C object file, filename.o, in a section after the VARS section as such:

```
<VARS   int     temp1,temp2;  >
< MAKEFILES     input.o >
```

In any MAKEFILES file, the first statement must be: #include "vars_defines.h" if any reference to a partname is made in the file. The input.c might be:

```
#include      "vars_defines.h"

input()
{
  printf("USER INPUT REQUIRED HERE:n");
  scanf("%d",&r0);
}
```

partname specified in the parts section of the HDL description is a valid C variable of type integer and storage class extern. Thus "r0" in the printf statement above is the part named R0 in our architecture.)

Now we will put all the parts together into a specification that is acceptable to AMISS. AMISS requires that certain keywords and tokens appear within the specification and that the parts of the specification be listed in a specific order. A formal description of the syntax acceptable to AMISS is found in appendix 2. It should suffice here to simply study the assembled

specification that we developed above to see how the parts fit together. A few points to note:

1. a specification starts off with a @@ BEGIN;

2. each section starts off with a @ followed by a AMISS keyword, as in "@ FORMATS";

3. the way in which the sections are ordered is as required by AMISS, thus the "@ PARTS" section must immediately follow "@ FORMATS" section;

4. comments are allowed only AFTER the "@@ BEGIN" statement; the grammar for a comment is that of the C programming language: /* comment here */;

5. AMISS does not care about white space;

6. the specification is all UPPER CASE except for setting constants in hex in the "@ INITIALIZATIONS" section, and declarations occurring within the "@ CFUNCTIONS" section.

7. IMPORTANT: we included a part named MPC.

```
@@ BEGIN
        /*      this is the description of
                a simple processor based
                on one described by Tannenbaum     */

        /*      this is an optional section        */
@ CFUNCTIONS
        <VARS   int temp,temp2; >

        /* here is a format description */

@ FORMATS
        FORMAT1:        <2> = OPCODE,
                        <3> = ADDRS,
                        <3> = ACNTRL,
                        <3> = BCNTRL,
                        <3> = CCNTRL,
                        <1> = ENC,
                        <1> = WRITE,
                        <1> = READ,
                        <2> = ALUF,
```

```
                        <1> = AMUXCNTRL,
                        <1> = MARCNTRL,
                        <1> = MBRCNTRL,
                        <2> = SHCNTRL,
                        <2> = COND;
@ PARTS
        <16> = AC, R1, R2, PC, AMASK, CONE, IR, TIR,
               ALATCH,BLATCH,
               ALUPORT1, ALUPORT2, ALU, MBR, MAR, AMUX,
               SHIFTER, ABUS, BBUS, CBUS;
        <8>  = MPC;
        <1>  = NBIT,ZBIT,VBIT,CBIT;
@ MEMORY
        <256><31> = CMEM;
        <1024><16> = MEM;
@ INITIALIZATIONS
        CONE = 1;
```

2. THE REGISTER TRANSFERS

So far we have laid down the groundwork for our simulator by defining its parts and its control memory format. Our next task is to describe the actions of the hardware. That is, we must define in the form of a program exactly what register transfers will occur and when they will occur with respect to one another. This, of course, must be done within the limitations of the control word format that we have established. Also, it must be done within the rules of using AMISS. For this reason, it is helpful here to consider now the AMISS system interfaces with our hardware simulator.

Real processors are clock dependent devices. A single clock cycle is broken up into equal time divisions and the processor is able to carry out certain functions during each of the divisions. For two reasons, an important keyword in the AMISS grammar is SUBCYCLE. The first reason is that we will wish to break up the functioning of our processor into subcycles and to allow only certain things to occur during each of the subcycles. Thus, our hardware program will contain statements like

```
IF ( SUBCYCLE EQ 1 ) {
        ....
      do something
        ...
}
```

The second reason that the use of the keyword SUBCYCLE is important is because the AMISS system itself utilizes it in generating the simulator and the program that provides the debugger user interface. So, at this point, let us decide that our processor will be based on a clock that is divided into 3 SUB-CYCLES.

FIGURE 2.1 contains the diagram of our processor and a picture of our control word format. Referring to FIGURE 2.1, we will now consider how our processor will work.

CDNO	SHCN	MBRC	MSRC	AMUX	ALUF	READ	WRIT	ENC	CCNT	BCNT	ACNT	ADDR	OPCD
2	2	1	1	1	2	1	1	1	3	3	3	8	2

Figure 2.1a. Control word format, numbers indicate field widths.

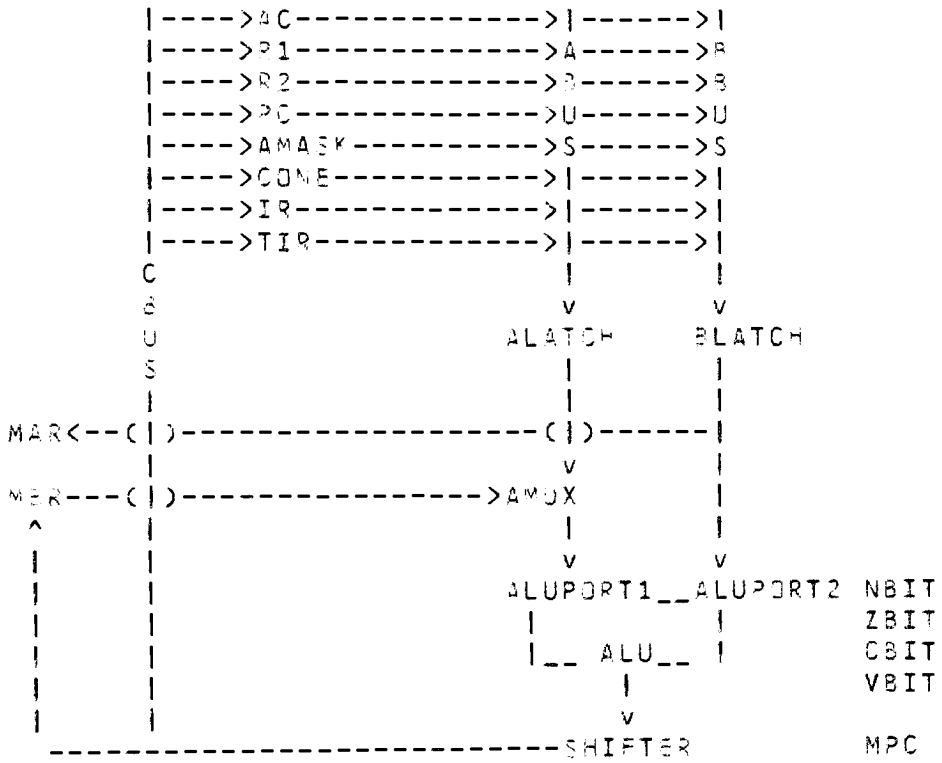


Figure 2.1b. The processor.

Let us decide that during the first SUBCYCLE data will be allowed to pass from any of the 8 registers within the register file, be latched by the two latches, and sent to the the MAR and the aluports via the ABUS and BBUS. Note the word allowed. What actually occurs depends upon the value of the fields in the control word. The fields to consider in controlling the feeding of the busses are the ACNTRL and BCNTRL. Thus, if ACNTRL contains a 2, that is, a binary 010, register R2 will be gated onto the ABUS. If BCNTRL contains a 5, a 101, register CONE will be gated onto the BBUS. This encoding could be described by a series of IF statements of the form:

```
IF ( ACNTRL EQ 0 ) ABUS = AC;
IF ( ACNTRL EQ 1 ) ABUS = R1;
        .
        .
        .
IF ( ACNTRL EQ 7 ) ABUS = TIR;
```

A second alternative would be to use the AMISS switch statement:

```
SWITCH ( ACNTRL) {
        CASE 0: ABUS = AC; BREAK;
        CASE 1: ABUS = R1; BREAK;
                ...
        CASE 7: ABUS = TIR; BREAK;
}
```

A third choice is to use the AMISS SELECT1 statement:

```
SELECT1( ABUS, ACNTRL, 3:
        AC, R1, R2, PC, AMASK, ONE, IR, TIR);
```

Actually, the SELECT statement is converted to a switch statement by AMISS. While the reason for using SELECTS might not be apparent here, it probably would be if our processor had many more elements or if it were horizontally encoded. In these cases, switch statements could be quite long and tedious to write. The data on the busses is next latched by the ALATCH and BLATCH. This can be described simply by:

```
ALATCH = ABUS;
BLATCH = BBUS;
```

Next, we will test the 1 bit MARCNTRL field to determine if the MAR is to be fed from the BBUS. Also, we will test the 1 bit AMUXCNTRL to determine if the output of the the AMUX will be its MBR input or its ALATCH input:

```
IF ( MARCNTRL EQ 1 ) MAR = BBUS;
IF ( AMUXCNTRL EQ 1 ) AMUX = MBR; ELSE AMUX = ALATCH;
```

Next we will set the the aluports to data on their inputs:

```
ALUPORT1 = AMUX;
ALUPORT2 = BLATCH;
```

At this point, in the form of a program, all the register transfers that we had decided to allow during the first subcycle have been tested for. Now we will enclose the program segment within a final enabling statement that ensures that this portion of our hardware is executed only when it is supposed to be. We will use the keyword SUBCYCLE to

do this:

```
IF ( SUBCYCLE EQ 1 ) {

        SELECT1( ABUS, ACNTRL, 8:
                AC, R1, R2, PC, AMASK, ONE, IR, TIR ):

        SELECT1( BBUS, BCNTRL, 8:
                AC, R1, R2, PC, AMASK, ONE, IR, TIR ):

        ALATCH = ABUS;
        BLATCH = BBUS;

        IF ( MARCNTRL EQ 1 ) MAR = BBUS;
        IF ( AMUXCNTRL EQ 1) AMUX = MBR; ELSE AMUX = ALATCH;

        ALUPORT1 = AMUX;
        ALUPORT2 = BLATCH;

}       /*      end of subcycle1 register transfers  */
```

Now we will consider the activities of the second subcycle. In SUBCYCLE 2 we will: allow the ALU to act on its inputs; set the status bits NBIT, ZBIT, CBIT, and VBIT according to the result of the ALU, send the ALU output to the SHIFTER and allow the SHIFTER to shift. First we must assign functions to the ALU. Since we have allowed 2 bits in the control field ALUF, we can define 4 functions. We will allow the ALU to add, to bitwise AND, to send the ALUPORT1 through, and to complement ALUPORT1. When the processor exerts an ALU function according to the control field ALUF, the element ALU will contain the result. The encoding of the control field ALUF will be:

```
        00      ALU = ALUPORT1 + ALUPORT2
        01      ALU = ALUPORT1 AND ALUPORT2
        10      ALU = ALUPORT1
        11      ALU = - ALUPORT1
```

We will use a SWITCH statement to describe the ALU functions:

```
SWITCH ( ALUF ) {
        CASE 0: ALU = ALUPORT1 + ALUPORT2; BREAK;
        CASE 1: ALU = ALUPORT1 AND ALUPORT2; BREAK;
        CASE 2: ALU = ALUPORT1; BREAK;
        CASE 3: ALU = - ALUPORT1; BREAK;
}
```

Now we will consider how to set the four status bits, NBIT, ZBIT, CBIT, and VBIT. First, some definitions of how the status bits will work in out hardware will be helpful. Remember that our ALU is 16 bits wide. We will number its bits from 0 to 15. The NBIT is to be set to 1 if the result of an alu function is negative. The ZBIT is set if the results is zero. The CBIT will be set if the result yields an overflow out of bit 15 of the ALU. The VBIT is set if a carry occurs from bit 14 into bit 15 of the ALU. This will occur only when the data in ALUPORT1 and ALUPORT2 have the same sign and an addition results in data that cannot be represented in 16 bits. Remember, since all AMISS machines use 2's complement representation, a carry from bit 14 into bit 15 in a 16 bit alu represents a sign change. In setting the status bits, we will make use of the two C variables that we declared in our specifications, TEMP, and TEMP2. The first thing to do is to reset all status bits to thier default value,zero.

```
NBIT = 0; ZBIT = 0; VBIT = 0; CBIT = 0;
```

Next, we will set variable TEMP to ALU. (NOTE: although we declared the variables in the CFUNCTIONS specifications in lower case letters, when we use them in the hardware description we must convert them to upper-case.)

```
TEMP = ALU;
```

To test whetner the status bits should be set, we will use the C language bitwise and operator: &. (We could have specified AND instead of &, they are the same.) To test if the NBIT is to be set, we will & the variable TEMP with the hex constant 0x8000. This will mask all but the sign bit. If the result is not zero, then the sign bit is set in the ALU and we will set the NBIT.

```
TEMP2 = TEMP & 0x8000;
IF ( TEMP2 NE 0 ) NBIT = 1;
```

To test if the CBIT is to be set, we will mask out the lower 16 bits of the ALU. If the result is not zero, then a carry must have occurred out of bit 15.

```
TEMP2 = TEMP & 0xffff0000;
IF ( TEMP2 NE 0 ) CBIT = 1;
```

To test if the ZBIT we will mask with 0xffff.

```
TEMP2 = TEMP & 0xffff;
IF ( TEMP EQ 0) ZBIT = 1;
```

We can test the for the VBIT only by using the values that were originally input to the ALU. If they were of the same sign and the ALU is of a different sign, then overflow has occurred. We will use the C boolean and operator &&. (We could have used BAND.)

```
IF ((ALUPORT1 GT 0) && (ALUPORT2 GT 0))
        IF ( NBIT NE 0 ) VBIT = 1;
IF ((ALUPORT1 LT 0) && (ALUPORT2 LT 0))
        IF ( ZBIT NE 1 ) VBIT = 1;
```

There are certainly other ways to write the code to determine how to set the status bits, including adding more parts to the hardware rather than using temporary variables. Now we will feed the SHIFTER from the ALU and allow the SHIFTER to shift. This will end SUBCYCLE 2.

```
SHIFTER = ALU;
SWITCH ( SHCNTRL ) {
        CASE 1: SHIFTER SL 1; BREAK;
        CASE 2: SHIFTER SR 1; BREAK;
        DEFAULT; BREAK;
}
```

Note that we used only three of the bits set aside for field SHCNTRL. One case, the DEFAULT, represents no shift. "SR" is a keyword representing SHIFTRIGHT (the C equivalent is >>=), and "SL" is SHIFTLEFT (<<=). SHIFTER SR 1 shifts the data in the SHIFTER one bit to the right. (There is no way to directly grab that shifted out bit in AMISS, although we could write some code to do so.) We will enclose our register transfers for subcycle 2 inside an IF statement and the subcycle will be:

```
F ( SUBCYCLE EQ 2) {

  SWITCH ( ALUF ) {
        CASE 0: ALU = ALUPORT1 + ALUPORT2; BREAK;
        CASE 1: ALU = ALUPORT1 AND ALUPORT2; BREAK;
        CASE 2: ALU = ALUPORT1; BREAK;
```

```
            CASE 3: ALU = -ALU; BREAK;
    }

    NBIT = 0; ZBIT = 0; VBIT = 0; CBIT = 0;

    TEMP = ALU;


    TEMP2 = TEMP & 0x8000;
    IF ( TEMP2 NE 0 ) NBIT = 1;


    TEMP2 = TEMP & 0xffff0000;
    IF ( TEMP2 NE 0 ) CBIT = 1;


    TEMP2 = TEMP & 0xffff;
    IF ( TEMP EQ 0) ZBIT = 1;


    IF ((ALUPORT1 GT 0) AND (ALUPORT2 GT 0))
            IF ( NBIT NE 0 ) VBIT = 1;
    IF ((ALUPORT1 LT 0) AND (ALUPORT2 LT 0))
            IF ( ZBIT NE 1 ) VBIT = 1;


    SHIFTER = ALU;

    SWITCH ( SHCNTRL ) {
            CASE 1: SHIFTER SL 1; BREAK;
            CASE 2: SHIFTER SR 1; BREAK;
            DEFAULT; BREAK;
    }
}   /* end subcycle 2 */
```

During the third subcycle we will allow the following: sending the output of the SHIFTER onto the CBUS; feeding any of the 8 registers in the register file from the CBUS according to the value of the CCNTRL field if the enable field ENC is set; feeding the MBR from the CBUS if the MBRCNTRL is set; reading and writing main memory according to the READ and WRITE fields; and setting the microprogramcounter to its new value depending upon whether a condition encoded in the COND field is met.

The first three of register transfers show nothing new except that we use a SELECT2. The SELECT2 works like the SELECT1 used previously except that the parts listed after the ':' are fed from the first named

element in the statement. Thus, translate the SELECT2 used below as:

```
SWITCH (CONTRL) {
        CASE 0: AC = CBUS; BREAK;
        CASE 1: R1 = CBUS; BREAK;
        ...
        CASE 7: TIR = CBUS; BREAL;
}
```

Our code then will be:

```
CBUS = SHIFTER;
IF (ENC EQ 1) {
    SELECT2( CBUS, CONTRL, 8:
    AC, R1, R2, PC, AMASK, CONE, IR, TIR);
}
IF (MBRCNTRL EQ 1) MBR = CBUS;
```

The memory read and write statement must use the AMISS keyword MEM and since memory is treated as an array of integers in hexidecimal representation, to access it requires that a statement specify an array element.

```
IF (READ) MBR = MEMEMAR];
IF (WRITE) MEMEMAR] = MBR;
```

Finally, the last thing that must be done in a description is to set the MPC (microprogram counter) to its new value. It is required that the keyword MPC be used and that the MPC be changed as the last thing done in any complete cycle. The reason for this is that AMISS interprets the attempt to test a control word field to apply to the control word pointed to by the current value of the MPC. If the MPC is changed prior to the end of the hardware description, and then an attempt is made to test a control field, the correct word will likely not be tested. We will set the MPC according to the two status bits NSIT and ISIT. Our control field is COND, and the field containing the next address if COND is met is ADDRF.

```
SWITCH (COND) {
   CASE 0:  MPC = MPC + 1; BREAK;
   CASE 1:  IF ( NZIT EQ 1) MPC = ADDRF;
                    ELSE MPC = MPC + 1; BREAK;
   CASE 2:  IF ( ZBIT EQ 1) MPC = ADDRF;
                    ELSE MPC = MPC + 1; BREAK;
   CASE 3:  MPC = ADDRF; BREAK;
}
```

Note here that in CASE 1 and CASE 2 an alternative to the IF has been provided by "ELSE MPC = MPC + 1;". Failure to provide for this is an easy thing to overlook and can be a debugging nightmare. If this is not done, the MPC won't be changed and the same control memory word will be tested again in the simulator.

We now can enclose the register transfers for SUBCYCLE 3 inside an IF statement, and assemble the three SUBCYCLES into one program. To the start of this program we preface the AMISS keywords @ MICROENGINE and a begin token "{"; to the end we append an end token "}" and the kewords "@@ END". Now we can append the the entire hardware description onto our original specification and we have described the processor so that AMISS can simulate it. The assembled program is as follows:

```
@@ BEGIN
@ CFUNCTIONS
        <VARS   int temp,temp2; >
@ FORMATS
        FORMAT1:        <2> = OPCODE,
                        <8> = ADDRE,
                        <3> = ACNTRL,
                        <3> = BCNTRL,
                        <3> = CCNTRL,
                        <1> = ENC,
                        <1> = WRITE,
                        <1> = READ,
                        <2> = ALUF,
                        <1> = AMUXCNTRL,
                        <1> = MARCNTRL,
                        <1> = MBRCNTRL,
                        <2> = SHCNTRL,
                        <2> = COND;
@ PARTS
        <16> = AC, R1, R2, PC, AMASK, CONE, IR, TIR,
               ALATCH, BLATCH,
               ALUPORT1, ALUPORT2, ALU, MBR, MAR, AMUX,
               SHIFTER, ABUS, BBUS, CBUS;
        <8>  = MPC;
        <1>  = NBIT,ZBIT,VBIT,CBIT;
@ MEMORY
        <128><31> = CMEM;
        <1024><16> = MEM;
@ INITIALIZATIONS
        CONE = 1;
        AMASK = 0xfff;

@ MICROENGINE
{
   IF ( SUBCYCLE EQ 1) {

        SELECT1(ABUS,ACNTRL,3: AC,R1,R2,PC,AMASK,CONE,IR,TIR);

        SELECT1(BBUS,BCNTRL,2: AC,R1,R2,PC,AMASK,CONE,IR,TIR);

        ALATCH = ABUS;
        BLATCH = BBUS;
        IF ( MARCNTRL EQ 1) MAR = BLATCH;
        IF ( AMUXCNTRL EQ 1 ) AMUX = MBR; ELSE AMUX  = ALATCH;

        ALUPORT1 = AMUX;
        ALUPORT2 = BLATCH;
   }

   IF ( SUBCYCLE EQ 2) {

        SWITCH (ALUF) {
                CASE 0: ALU = ALUPORT1 + ALUPORT2; BREAK;
                CASE 1: ALU = ALUPORT1 AND ALUPORT2; BREAK;
```

```
                CASE 2: ALU = ALUPORT1; BREAK;
                CASE 3: ALU = - ALUPORT1; BREAK;
        }

        NBIT = 0; ZBIT = 0; VBIT = 0; CBIT = 0;
        TEMP = ALU;

        TEMP2 = TEMP & 0x8000;
        IF ( TEMP2 NE 0 ) NBIT = 1;

        TEMP2 = TEMP & 0xffff0000;
        IF ( TEMP2 NE 0 ) CBIT = 1;

        TEMP2 = TEMP & 0xffff;
        IF ( TEMP2 EQ 0) ZBIT = 1;

        IF ((ALUPORT1 GT 0) && (ALUPORT2 GT 0))
                IF ( NBIT NE  0 ) VBIT = 1;

        IF ((ALUPORT1 LT 0) && (ALUPORT2 LT 0))
                IF ( NBIT NE 1 ) VBIT = 1;

        SHIFTER = ALU;

        SWITCH (SHCNTRL) {
                CASE 1: SHIFTER SL 1; BREAK;
                CASE 2: SHIFTER SR 1; BREAK;
                DEFAULT: BREAK;
        }
    }

    IF ( SUBCYCLE EQ 3) {

        CBUS = SHIFTER;

        IF (ENC EQ 1) {
           SELECT2(CBUS,CONTRL,8: AC,R1,R2,PC,AMASK,CONE,IR,TIR);
        }

        IF (MBRCNTRL EQ 1) MBR = CBUS;

        IF (READ) MBR = MEM[MAR];
        IF (WRITE) MEM[MAR] = MBR;

        SWITCH ( COND ) {
                CASE 0: MPC = MPC + 1; BREAK;
                CASE 1: IF ( NBIT EQ 1) MPC = ADDRF;
                        ELSE MPC = MPC + 1; BREAK;
                CASE 2: IF ( ZBIT EQ 1) MPC = ADDRF;
                        ELSE MPC = MPC + 1; BREAK;
                CASE 3: MPC = ADDRF; BREAK;
        }
    }
}
```

@@ END

A few comments regarding design may be helpful. We discussed the procedure for specifying the control word format in the case that we had more than one format. Let us say that we have a processor that uses two formats: one for opening gates, and one used for branching. This would be encoded in the OPCODE field. In our register transfer code, we would want to first do a test of the OPCODE before performing any register transfers. Our code might look like:

```
@ MICROENGINE
{
        IF (OPCODE EQ 0) {
            IF (SUBCYCLE EQ 1) {
                    ...
            }
            IF (SUBCYCLE EQ 2) {
                    ...
            }
        }

        IF (OPCODE EQ 1) {
            IF (SUBCYCLE EQ 1) {
                    ...
            }
            IF (SUBCYCLE EQ 2) {
                    ...
            }
        }
}
```

In such a case, it is suggested that both program segments test the same number of SUBCYCLES, even if nothing occurs during a SUBCYCLE. We could write an empty SUBCYCLE by:

```
        IF (SUBCYCLE EQ 1) {
        }
```

or

```
        IF (SUBCYCLE EQ 1)
                ;
```

Although AMISS won't balk if this is not done, it will cause the debugger interface to be inconsistent. AMISS does require that the last SUBCYLE be explicitly tested for in each program segment.

The AMISS program that parses the user program is called RRUN. If the program that we wrote above exists in a file named EXAMPLE, we can submit it to RRUN by:

$RRUN EXAMPLE.

If EXAMPLE has no syntax errors, RRUN will parse it, accept it, write a number of files, and perform a "make" on those files resulting in an executable simulator named CPU. This is all automatic. The files other than CPU and RR.nu (a copy of EXAMPLE) are moved to a directory named RRdir. It is up to the user to remove them. CPU needs two files to work. The first is the file cmemory. This is a file of integers written in decimal corresponding to the simulated control memory. The second file is called memory. This is another file of integers written in hexidecimal. CPU uses one more file, if it is provided. This file is named instrfile and is a system file containing the mnemonic form of a microprogram. To execute CPU one types

$CPU [-options]

CPU with no options will run CPU. In this case, the only output that appears is an indication of how many microinstructions were executed before the processor was halted by a halt statement. The two options are:

$CPU -h and $CPU -d

For each microinstruction, the -h option gives output consisting of the mnemonic form of the microinstruction, a header with the fields of the control format and the value of each field, and a count of the number of the microinstruction that have been executed. The -d option provides a debugger interface to the executing simulator. The manual describing the debugger is in Appendix 4.

If EXAMPLE contained a syntax error, RRUN would report the line number of EXAMPLE in which the error occurred. In this case, EXAMPLE would have to be modified and resubmitted to RRUN.

3. THE MICROASSEMBLER

This section discusses the use of MPL, the part of the AMISS package that assists the designer in writing a microassembler. We will use the processor described above and write a microassembler in a high level register transfer language based on that described by Tannenbaum [6]. We will also write a microprogram to interpret a few instructions based on Tannenbaum's instruction set for a stack oriented machine and submit it to our microassembler.

First, let us discuss a few details concerning how MPL works. MPL provides an interface to a UNIX utility program called YACC [12]. YACC is a very useful program that can be used to write language recognizers and to execute C language statements when elements of the language are recognized. In order to write the language recognizer, YACC requires that the user provide, among other details, the grammer of his language and its actions (the C statements). MPL attempts to assist the designer in providing these, and to save the him the trouble of having to provide most of the other details that YACC requires. Because MPL interfaces with YACC, the rules for using MPL reflect those of YACC. YACC requires that the languge be described by first declaring keywords called tokens. Next, a list of rules that define the order in which the tokens may be combined into sentences of the language are provided. Finally, actions to be carried out upon recoginizing a sentence are specified as C language statements. MPL provides a built in list of tokens and appends to them the names of the microarchitecture elements listed under the PARTS section of the hardware description. The tokens available for developing the language are:

```
IF THEN GOTO N Z EQ
LSHIFT RSHIFT
BAND INV RD WR ER AND OR EOR BOR
```

In addition, MPL provides a selection of single character tokens that may be included in the grammer rules. These are:

```
( ) + - = ,
0 1 2 3 4
5 6 7 8 9
```

Whenever one of these single character tokens are used in a grammar rule, it must be surrounded by quotations marks. For example:

```
EQ "=" EQ "+" "1"
```

or

```
IF "(" IN EQ "1" ")"
```

The input to MPL that must be provide by the designer is a list of the rules that describe the acceptable sentences of the high level register transfer language and a specification of how the fields of a control word are to be set when a sentence is recognized.

First let us detail our macro instruction set. We will work with three instructions among those described by Tannenbaum, STOD, ADD, LOCO, and add an additional one, HALT. (FIGURE 3.1)

Binary	Mnemonic	Instruction	Meaning
0001xxxxxxxxxxx	STOD	store direct	mem[x] := ac
0010xxxxxxxxxxx	ADDD	add direct	ac:=ac+mem[x]
0111xxxxxxxxxxx	LOCO	load constant	ac := x
1111000000000000	HALT	halt processor	

FIGURE 3.1 Instructions adapted from Tannenbaum [12].
The first four bits are the opcode.
xx...xx in the Binary column
corresponds to x in the Meaning column.

Although this instruction set is small, it will serve for the purpose of example. Also, one of the advantages of MPL is that new instructions can be added easily.

Every useful microprogram must have a facility for branching to different addresses in control memory. MPL requires that the first part of its input be a description of the grammar that the designer wishes to use for branching. This description must be listed after the MPL keyword MMTEST. In our language, we will arrange the following keywords among those available in MPL :

IF, N, Z, GOTO

Three other keywords must be used in the branch description, LABELOPERAND, MMADDRESS, and OPCODE. LABELOPERAND can be thought of as a pseudonym for any label in a microprogram. Thus, if a statement in a program were "GOTO FETCH", MPL would accept "FETCH" to mean LABELOPERAND. MMADDRESS is used in actions. It is used by MPL to stand for an address in control memory. You will remember that we used a field named ADDRF in our control word format to hold the address of a word in control memory. Thus in our actions, we might set "ADDRF = MMADDRESS;". Finally, remember that every control word format must have as its first field one that is two bits wide named OPCODE. It follows from this that every microinstruction must have the OPCODE field set. In order to ensure this, the action section of a grammar rule must start off by setting OPCODE. MPL also requires that the definition for describing the branching grammar follow a particular syntax. The formal definition of this syntax is in Appendix 2. By illustration, the syntax is:

```
 DEFINE TEST {
%%
MMTEST:
     " ... define test
     instruction grammar RULE 1 ..."  { OPCODE = "a number";
                                       "some field" = MMADDRESS;
                                       "set some other fields";}

   | "... define test
     instruction grammar RULE 2 ..."  { OPCODE = "a number";
                                       "some field" = MMADDRESS;
                                       "set some other fields";}

    | "... define test
      instruction grammar RULE 3 ..."  { OPCODE = "a number";
                                       "some field" = MMADDRESS;
                                       "set some other fields"; }


    ;
%%
}
```

Note in the illustration that individual RULES are separated by the marker "|" and that the entire definition is ended by a ";". (These are YACC syntax requirements.) Think of the entire specification as being a listing of three alternatives to defining a sentence named MMTEST.

We will use N (a keyword) to correspond to the same part we called NBIT in our processor, and Z to correspond to ZBIT. The branch portion of our high level microprogramming language will be:

```
DEFINE TEST {
%%
MMTEST:   IF N GOTO LABELOPERAND     {OPCODE = 0;
                                      ADDRF = MMADDRESS;
                                      COND = 1;
                                     }
        | IF Z GOTO LABELOPERAND     {OPCODE = 0;
                                      ADDRF = MMADDRESS;
                                      COND = 2;
                                     }
        | GOTO LABELOPERAND          {OPCODE = 0;
                                      ADDRF = MMADDRESS;
                                      COND = 3;
                                     }
%%
}
```

Note in the actions part above that the setting of the control field COND corresponds to the way that we decided to test for the next MPC address in our hardware description.

The next step in defining our high level language is to specify all the other statements from which it will be made and to associate actions with each of them. One type of statement that will likely be common to all languages is one in which one element of the architecture, probably a register, is assigned the value of another. Our processor has 8 registers in its register file. Consider the statement "AC = IR". Let us describe one set of actions that will accomplish this :

```
ACNTRL  = 6     put IR on the ABUS
ALUF    = 2     send ALUPORT1 thru ALU
ENC     = 1     enable CBUS
CONTRL  = 0     load AC from CBUS

all other control fields are set to 0
```

In order to recognize all the possible assignment statements possible with 8 registers would require 64 statements of the form:

```
    |   AC "=" IR                {OPCODE = 0;
                                  ACNTRL = 6;
                                  ALUF = 2;
                                  ENC = 1;
                                  CCNTRL = 0;
                                 }
```

Note that setting AC to the other 7 registers would require rules that differed only in one value in their actions, the encoding of the ACNTRL. To avoid having to do this, MPL will accept a "DEFINE SEMANTICS" statement that takes a short description and turns it into a number of rules. This statement accepts a listing of field names and constants to which the fields are to be set in each rule. Next, it interprets a specification of how to set the single value that differs among all the actions. Thus, to specify setting AC to the value of any of the other 7 registers in our processor we would write:

```
|   DEFINE SEMANTICS {
      CONSTANTS: OPCODE = 0, ALUF = 2, ENC = 1, CCNTRL = 0;
      SELECT( AC, ACNTRL, 8:
            AC,R1,R2,PC,AMASK,CONE,IR,TIR)
    }
```

This is rewritten by MPL as 8 rules differing only in the value to which ACNTRL is to be set when the rule is recognized. The "8" in the SELECT part causes 8 statements to be added to the language and sets the ACNTRL to 0 for the first one, to 1 for the second one, to 2 for the third one, and so on. It is essential that the number match the number of parts listed and that the parts are listed in the order that will match the way we wish to evaluate ACNTRL (from ACNTRL = 0 to ACNTRL = 7). The 8 statements are: AC = AC, AC = R1, AC = R2, AC = PC, AC = AMASK, AC = CONE, AC = IR, and AC = TIR. Thus, the following will result from the above DEFINE SEMANTICS statement:

```
        |   AC "=" AC     { OPCODE = 0;
                            ALUF = 2;
                            ENC = 1;
                            CCNTRL = 0;
                            ACNTRL = 0;
                          }
        |   AC "=" R1       { OPCODE = 0;
                            ALUF = 2;
```

```
                        END = 1;
                        CONTRL = 0;
                        ACNTRL = 1;
                    }
            ...

    |  AC "=" TIR    { OPCODE = 0;
                          ...
                        ACNTRL = 7;
                    }
```

Our high level language specification will have 8 DEFINE SEMANTICS statements corresponding to the 8 registers in the register file.

Now we must explicitly define all of the rest of the statements that are part of the language as well as their associated actions. The words from which sentences of our language may be made include the names of any of the parts that were specified as elements of the microarchitecture and the keywords provided by MPL. The MPL keywords are:

```
IF THEN N Z GOTO EQ TEST BIT
LSHIFT RSHIFT BAND INV RD WR
BR AND OR EOR BOR
```

Because our processor is based on one described by Tannenbaum [6], in the remainder of this example we will develop part of the language that he provides. The following statements will be added to our language:

```
ALU = TIR, MAR = PC, PC = PC + 1,
IR = MBR, TIR = LSHIFT(IR + IR),
TIR = LSHIFT(TIR), TIR = RSHIFT(TIR),
TIR = BAND(IR,AMASK),
MAR = IR, MBR = AC, AC = MBR + AC,
RD, WR
```

Figure 3.2 is a table that indicates how the control fields should be set for each of the instructions.

	C O N D	S H I F T	M B R	M A R	A M U X	A L U	A L U F	R A M D	W R I T E	C O N D	C O N D	A D D R	A L U N	D O N E
MAR=PC							1					3		
ALU=TIR						2						7		
PC=PC+1						0			1	3	5	3		
IR=MBR					1				1		6			
TIR=LSHIFT(TIR)		1				2			1	7		7		
TIR=RSHIFT(TIR)		2				2			1	7		7		
TIR=LSHIFT(IR+IR)		1				0			1	7	6	6		
TIR=BAND(IR,AMASK)						1			1	7	6	4		
MAR=IR				1							6			
MBR=AC			1			2						0		
AC=MBR+AC					1	0			1	0		0		
RD							1							
WR								1						

FIGURE 3.2 Setting of control fields for the example processor. All fields without numbers are set by MPL to 0.

A few points concerning Figure 2 are noteworthy. The statement "TIR = LSHIFT(IR + IR)" causes an addition of IR to itself which is a leftshift of the IR. This addition will set the status bits in our processor. After the addition, another leftshift occurs. Last, the IR leftshifted twice is stored in the TIR. All this occurs within one cycle. The statement "TIR = BAND(IR,AMASK)" causes a Boolean AND (true BAND) of IR and AMASK storing the result in TIR. A statement such as "TIR = TIR" must go through the ALU in our processor. It is useful in setting the status bits. The remainder of the microinstructions should require no explanation.

Now we will list the entire MPL description:

```
DEFINE TEST {
%%
MMTEST:  IF N GOTO LABELOPERAND {OPCODE = 0;
                                 ADDRF = MMADDRESS;
                                 COND = 1;
                                 }
         IF Z GOTO LABELOPERAND  {OPCODE = 0;
                                  ADDRF = MMADDRESS;
                                  COND = 2;
```

```
                                        }
|       GOTO LABELOPERAND                {OPCODE = 0;
                                         ADDRF = MMADDRESS;
                                         CONE = 3;
                                         }
;
%%
}

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 0;
          SELECT(AC, ACNTRL, 8:
                 AC, R1, R2, PC, AMASK, CONE, IR, TIR)
        }

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 1;
          SELECT(R1, ACNTRL, 8:
                 AC, R1, R2, PC, AMASK, CONE, IR, TIR)
        }

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 2;
          SELECT(R2, ACNTRL, 8:
                 AC, R1, R2, PC, AMASK, CONE, IR, TIR)
        }

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 3;
          SELECT(PC, ACNTRL, 8:
                 AC, R1, R2, PC, AMASK, CONE, IR, TIR)
        }

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 4;
          SELECT(AMASK, ACNTRL, 8:
                 AC, R1, R2, PC, AMASK, CONE, IR, TIR)
        }

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 5;
          SELECT(CONE, ACNTRL, 8:
                 AC, R1, R2, PC, AMASK, CONE, IR, TIR)
        }

|       DEFINE SEMANTICS {
          CONSTANTS: OPCODE = 0, ALUF = 2,
                     ENC = 1, CONTRL = 6;
```

```
  SELECT(IR, ACNTRL, 8:
        AC, R1, R2, PC, AMASK, CONE, IR, TIR)
}

DEFINE SEMANTICS {
  CONSTANTS: OPCODE = 0, ALUF = 2,
             ENC = 1, CONTRL = 7;
  SELECT(TIR, ACNTRL, 8:
        AC, R1, R2, PC, AMASK, CONE, IR, TIR)
}

ALU "=" TIR       {OPCODE = 0;
                   ACNTRL = 7;
                   }
MAR "=" PC        {OPCODE = 0;
                   BCNTRL = 3;
                   MARCNTRL = 1;
                   }
MAR "=" TIR       {OPCODE = 0;
                   BCNTRL = 7;
                   MARCNTRL = 1;
                   }
PC "=" PC "+" "1"          {OPCODE = 0;
                   ACNTRL = 3;
                   BCNTRL = 5;
                   ALUF = 0;
                   ENC = 1;
                   CONTRL = 3;
                   }
IR "=" MBR        {OPCODE = 0;
                   AMUXCNTRL = 1;
                   ALUF = 2;
                   ENC = 1;
                   CONTRL = 6;
                   }
MAR "=" IR        {OPCODE = 0;
                   BCNTRL = 6;
                   MARCNTRL = 1;
                   }
MBR "=" AC        {OPCODE = 0;
                   ACNTRL = 0;
                   ALUF =2;
                   MBRCNTRL = 1;
                   }
AC "=" MBR "+" AC          {OPCODE = 0;
                   BCNTRL = 0;
                   AMUXCNTRL = 1;
                   ALUF = 0;
                   ENC = 1;
                   CONTRL = 0;
                   }
TIR "=" LSHIFT"("TIR")"
                  {OPCODE = 0;
                   ACNTRL = 7;
```

```
                        ALUF = 2;
                        SHCNTRL = 1;
                        ENC = 1;
                        CCNTRL = 7;
                       }
|       TIR "=" RSHIFT"("TIR")"
                       {OPCODE = 0;
                        ACNTRL = 7;
                        ALUF = 2;
                        SHCNTRL = 2;
                        ENC = 1;
                        CCNTRL = 7;
                       }
|       TIR "=" LSHIFT"("IR"+"IR")"
                       {OPCODE = 0;
                        ACNTRL = 5;
                        BCNTRL = 6;
                        ALUF = 0;
                        SHCNTRL = 1;
                        ENC = 1;
                        CCNTRL = 7;
                       }
|       TIR "=" BAND"("IR",AMASK")"
                       {OPCODE = 0;
                        ACNTRL = 6;
                        BCNTRL = 4;
                        ALUF = 1;
                        ENC = 1;
                        CCNTRL = 7;
                       }
|       RD             {OPCODE = 0;
                        READ = 1;
                       }
|       WR             {OPCODE = 0;
                        WRITE = 1;
                       }
```

Now that we have a specification of our language, we can submit it to the MPL program MPLRUN which actually writes the microassembler. MPLRUN requires one other file, the original hardware description. In our example, we named it EXAMPLE. If our grammar rules were in a file named LANGUAGE, to build the microassembler we would type:

```
MPLRUN LANGUAGE EXAMPLE
```

MPLRUN would parse the LANGUAGE file and, if it was accepted, would write a number of system files. These files are automatically made into an executable microassembler named MICROASM. The files are then moved into a directory named MYdir and it is up to the

user to remove them. MICROASM itself is a YACC based parser that accepts a microprogram written in our language.

Let us now write a microprogram to interpret our small instructions set. (FIGURE 3.1 is duplicated below.)

Binary	Mnemonic	Instruction	Meaning
0001xxxxxxxxxxx	STOD	store direct	mem[x] := ac
0010xxxxxxxxxxx	ADDD	add direct	ac:=ac+mem[x]
0111xxxxxxxxxxx	LODD	load constant	ac := x
1111000000000000	HALT	halt processor	

FIGURE 3.1 Instructions adapted from Tannenbaum [12].
The first four bits are the opcode.
xx...xx in the Binary column corresponds to
x in the Meaning column.

Our strategy will be to fetch a macro instruction, decode the opcode by shifting and testing the NBIT in our processor, branch to a routine to perform the register transfers to execute the instruction, and then fetch the next macro instruction. We will halt the processor upon decoding the HALT instruction. All microprograms must meet certain syntax requirements:

1. All statements other than comments are upper case.

2. A microprogram must begin with the word "START:" and end with the word "END:", and both must be on lines with nothing else.

3. A label must start with a letter and end with a ":".

4. A line with a label on it must have the label as the first non-blank thing on the line.

5. A line with a label on it must also have a microinstruction on it.

6. A single microinstruction must be able to be written on one line and is ended by a ";".

7. If more than one statement makes up a single microinstruction, the individual statements are separated by ",".

8. Comments are enclosed between "@" and comments cannot extend beyond one line.

9. Blank lines and lines having only comments ARE accepted.

Here is a microprogram to interpret the 4 instructions:

```
START:
FETCH: MAR = PC, RD;      @ fetch an instruction @
       PC = PC + 1;
       IR = MBR, IF N GOTO HALT; @ if N then 1111 @

       TIR = LSHIFT(IR + IR), IF N GOTO LDCO; @ a 0111 @
       TIR = TIR, IF N GOTO ADDO; @ a 0010 @

       TIR = BAND(IR,AMASK);  @ a 0001 is a STO@
                                @ mask out opcode @
       MAR = TIR;               @ set mar to addr @
       MBR = AC, WR;          @ value in mbr & wr@
       GOTO FETCH;

ADDO:  TIR = BAND(IR,AMASK);  @ a 0010 is a ADDD@
       MAR = TIR, RD;          @ mask out opcode & rd@
       AC = MBR + AC, GOTO FETCH;

LDCO:    TIR = BAND(IR,AMASK); @ mask off opcode @
         AC = TIR,  GOTO FETCH;@ store in AC @

HALT:    GOTO END;
END:
```

If this program existed in a file named MCODE, we would submit it to the microassembler by

$MICROASM MCODE

If no syntax errors were detected, three files would result asmlist, cmemory, instrfile. asmlist is the listing file

containing the control memory word that was assembled from each microinstruction, the microinstruction itself, and a symbol table of addresses. The other two files are used by the simulator CPU.

The listing file for our microprogram is (comments have been deleted here because of space limitations) :

```
1                    START:

2     0 0 0 0 1 0 0 1 0 0 0 3 0 0 0 FETCH:        MAR = PC, RD;
3     1 0 0 0 0 0 0 0 0 1 3 5 3 0 0       PC = PC + 1;
4     2 1 0 0 0 1 2 0 0 1 6 0 0 f 0       IR = MBR, IF N GOTO END:
5     2
6     3 1 1 0 0 0 0 0 0 1 7 6 6 c 0       TIR = LSHIFT(IR + IR), IF N GOTO LOCO
7     4 1 1 0 0 0 2 0 0 1 7 0 7 9 0       TIR = TIR, IF N GOTO ADDO:
8     4
9     5 0 0 0 0 0 1 0 0 1 7 4 5 0 0       TIR = BAND(IR,AMASK);
10     6 0 0 0 1 0 0 0 0 0 0 7 0 0 0      MAR = TIR;
11     7 0 0 1 0 0 2 0 1 0 0 0 0 0 0      MBR = AC, WR;
12     8 3 0 0 0 0 0 0 0 0 0 0 0 0 0      GOTO FETCH;
13     8
14     9 0 0 0 0 0 1 0 0 1 7 4 6 0 0 ADDO:        TIR = BAND(IR,AMASK);
15    10 0 0 0 1 0 0 1 0 0 0 7 0 0 0      MAR = TIR, RD;
16    11 3 0 0 0 1 0 0 0 1 0 0 0 0 0      AC = MBR + AC, GOTO FETCH;
17    11
18    12 0 0 0 0 0 0 1 0 0 1 7 4 6 0 0 LOCO:    TIR = BAND(IR,AMASK);
19    13 3 0 0 0 0 0 2 0 0 1 0 0 7 0 0      AC = TIR,  GOTO FETCH;
20    13
21    14 3 0 0 0 0 0 0 0 0 0 0 0 0 f 0 HALT:    GOTO END;
22    15 END:
      ********SYMBOL TABLE********
      START 00000
      FETCH 00000
       ADDO 00009
       LOCO 00013
       HALT 00014
        END 00015
```

If we wanted to add to our macroinstruction set, all that would be required would be to submit the modified microoorogram to MICROASM. If it were necessary to add statements to the microassembler, we would have to modify the input to MPLRUN (the file we named LANGUAGE) and resubmit it.

APPENDIX 4: CPU DEBUGGER COMMANDS

This section contains a synopsis of the commands available in the debugger option of CPU.

NAME audit, a

SYNTAX audit on audit off

DESCRIPTION

Audit control writing of the transcript of the debugger session to the file RR.audit. Audit can be set on or off at any time.

NAME display, d

SYNTAX

	examples:
display [partname]	display mpc
display [partname] , [partname] ...	display mpc, pc
display mem [integer]	display mem fb4
	display mem D4200
display mem [integer]-[integer]	display mem fa0-fb4
	display mem D4000-D4200
display cmem [integer]	display cmem 64
	display cmem D100
display cmem [integer] - [integer]	display cmem a-f
	display cmem D10-D15

DESCRIPTION

Display is valid in both states 1 and 2. Display shows the contents of partname as " partname = number " where number is in hexidecimal. As many parts as may be listed on one line are displayed one line at a time. To display a memory address use "mem [integer]", where integer is an address. To display a range of memory addresses, use d [integer] - [integer], where integer-integer is the range of addresses to be displayed. To display a cmemory address or range of addresses, use cmem [integer], and cmem [integer] - [integer]. If partname does not exist an error is reported. If addresses are out of range an error is reported.

NAME mset, m

SYNTAX mset [integer] = [integer]

examples

```
mset fb4 = 64
mset fb4 = D100
mset D4200 = 64
mset D4200 = D100
```

mset [integer] - [integer]

```
mset fa0 - fb4
mset D4000 - D4020
```

DESCRIPTION

Mset allows the setting of main memory addresses. In the command "mset [integer] = [integer]", the first integer is the address and the second is its new value. Integer is hexidecimal by default. If integer is prefaced by the letter D, it is decimal. In the command "mset [integer] - [integer]", the integers mark the beginning and end of a range of consecutive memory addresses to be set. In this case, a prompt containing the address is presented, after which the new value for that address is to be input. If any address is out of range, an error is reported.

NAME quit, q

SYNTAX quit

DESCRIPTION

Quit is valid in both states 1 and 2. In state 1, quit leaves the execution of CPU. In state 2, quit causes the processor simulation to stop and returns the debugger to state 1.

NAME reset

SYNTAX reset

DESCRIPTION

Reset is valid in both states 1 and 2. Reset sets all the elements of the processor to their original values, ie. the values at the start of the simulation. Memory and cmemory are not reset.

NAME return

SYNTAX "hit return key"

DESCRIPTION

The return key is a valid command in state 2. If the simulation is running in a trace debug state, the return key controls the continuation of the simulation after the "ENTER2:" promot. In state 1, after a "ENTER1:" promot, the return key causes a new "ENTER1:" promot to be issued. Other than this, the return key has no affect in state 1.

NAME run, r

SYNTAX run
run [integer]

DESCRIPTION

Run is valid only in state 1. Run causes the debugger to initiate a simulation of the processor. Run with no argument runs the processor until the processor is halted by decoding an instruction that causes the processor to halt or until 10000 cycles have been executed. Run integer will run the number of cycles indicated by integer. Integer is decimal. If an instruction that halts the processor is executed before number cycle have been executed, the processor will halt and report this. The output that appears after initiating a run command depends on how the debugger was last set using the trace command.

SEE ALSO: trace.

NAME set, s

SYNTAX

	examples
set [part] = integer	set mpc = a set mpc = D10
set [part] = integer, [part] = integer ...	set mpc = a, pc = D10

DESCRIPTION

The set command is valid in both states 1 and 2. A valid part of the architecture is set to some value, number. Number is in hexidecimal notation. If number is not representable in the number of bits that partname contains, number is truncated to fit. If partname is not a part of the architecture, an error is reported. If a list of partnames is given and one or more of the partnames is not a part of the architecture, all of the valid parts listed before the first invalid partname are set. As many parts as may be listed on one line separated by "," are set.

NAME trace, t

SYNTAX
```
trace off       t f
trace on        t o
trace debug     t d
trace header    t h
```

DESCRIPTION

The trace command is valid in both states 1 and 2. Its affects are only seen after a simulation has been initiated after a run command. Trace sets the output of the simulated running of the processor according to:

trace off - no output;

trace on - output consists of a heading containing the control field names and values for the current contents of the microinstruction register; each part of the architecture is listed with its contents in hexidecimal; a report of the current SUBCYCLE and the number of cycles that have been executed since the last processor reset command in PASS is given; output is one screen full followed by the prompt "more"; carriage return continues output, any other key causes processor to halt and return to state 1;

trace debug - output is identical to trace on except user input is required after output is presented; user is prompted for input by "ENTER2:" which requires the input of any valid state 2 command.

trace header - output is identical to trace debug except the contents of the architecture parts are not displayed.

SEE ALSO: run

NAME vi, v

SYNTAX vi mem
vi cmem
vi hw

DESCRIPTION

vi is valid in both states 1 and 2. vi causes the UNIX visual editor to open the file memory, cmemory, or a copy of the user hardware input file.

BUGS: any changes made to memory or cmemory by vi are effected in the simulation BUT changes made by the executing simulator do NOT affect the files memory or cmemory. changes made to the hardware file have no affect on the original file.

SEE ALSO: UNIX online man vi.

APPENDIX 5. SUGGESTIONS FOR USING AMISS.

Some suggested steps in designing a microarchitecure.

1. Draw a box type diagram containing the name of the parts of the microarchitecture and the bus connections between the parts. Save the diagram for future reference.

2. Use partnames and control field names that are more tnat one letter long. Use names that are as small as possible while still retaining some mnemonic meaning.

3. Include only as many parts and connections as are useful in the simulation, ie. if you can reasonably describe the fact that the output of part A goes to part B by " B = A " don't put a bus between them named ABBUS that requires " ABBUS = A; B = ABBUS; ". On the other hand, don't throw out everything.

4. Vertically encoded control formats are easier and cleaner to describe than horizontal ones. Even if a control format is completely horizontal, it is probable that control points can be arranged in groups and described using a field with one name instead of many one bit fields.

5. Make use of the CFUNCTIONS variables option.

6. Use temporary variables, especially when testing for determining if a microprogram branch should be taken.

7. Keep a listing of the HDL hardware description handy when debugging. To look at the C version of this, see tne file "RRhardware.c". To look at it while in the debugger, use the command "vi hw". (But remember, editing this file will NOT affect the current simulation. If an error is found in the hardware file, you must quit the simulation, fix the error, and resubmit the HDL program to RRUN with or without the RRUN -h option.)

8. Don't forget to ensure that the MPC is affected somehow at the end of every complete cycle. Otherwise the simulator will keep using the same control word ad infinitum.

9. Divide the register transfers into subcycles by

```
IF (SUBCYCLE EQ number) {
        ....
}
```

Otherwise the debugger interface won't work.

Suggestions for using MPL:

1. Be very careful when describing how the fields of a control word should be set (ie. the actions).

2. Use the "DEFINE SEMANTICS" statement for every part in the register file. This will add the statements that any part in the register file can be set to any other part to the micro-language.

3. Don't forget to set OPCODE as the first thing done in the actions.

4. Remember that any single character in a grammar rule including digits are surrounded by quotation marks.

Suggestions for writing a microprogram:

1. The microprogram must start with keyword START and end with keyword END.

2. Ensure that the last thing that is done after a series of microinstructions have decoded and executed a macroinstruction is to return control to the fetch portion of the microprogram.

3. Include some kind of HALT instruction in the macro instruction set and when it is decoded by the microprogram, cause the processor to halt using a "GOTO END" statement.

APPENDIX 6 AMISS USER'S GUIDE

This section contains a guide to using AMISS.

1. To submit an HDL program to RRUN:

```
%RRUN <filename>
```

2. To resubmit an HDL program to RRUN that differs from the last submitted one only in the register transfer section (ie. NOT the specifications section):

```
%RRUN -h <filename>
```

3. If the files cmemory, memory, and optionally instrfile are in the same directory, to run the simulation:

```
%CPU          gives notice when procesor halts
%CPU -h       gives header output for each
              microinstruction
%CPU -d       enters the simulation debugger
```

4. To submit a language description to the microprogramming generator MPL:

```
%MPLRUN <file1> <file2>
```

where file1 is the language description and file2 is an HDL hardware program that has already passed a syntax check by RRUN.

5. To submit a microprogram to MICROASM, the microassembler generated by MPLRUN:

```
%MICROASM <file1>
```

where file1 is a microprogram. The listing output is in the file named "asmlist", the control memory is in the file named "cmemory".

AMISS ERROR REPORTS

Errors reported by AMISS are designed to be self explanatory. Error reports for the Hardware Section and the Microprogramming Language Section are discussed here.

1) The Hardware Section.

If a syntax error is found in an HDL program submitted to RRUN, the line number on which the error occurred and the character or name that caused the parser to stop is reported. Parsing stops after the first syntax error.

The debugger error statements are:

1. "BAD COMMAND" indicates that the command is not available.

2. "PART <name> does not exist" indicates that the part named in a set or display command is not a part of the microarchitecture

3. "MEMORY <integer> (0<integer>)
   OUT OF RANGE
   MAXIMUM ADDRESS = <integer>"

   indicates that in a display or meet command, an attempt was made to access either a memory or control memory address greater than the maximum specified in the HDL memory specification.

4. "INVALID OPTION" indicates that in a trace or audit command, an invalid option was specified

5. "NOT VALID HERE" indicates that while in a simulation after an ENTER2: prompt, an attempt was made to enter a run command

2). The Microassembler Section.

If a syntax error is found in the LANGUAGE description file input to MPLRUN, the line number and the character or name that caused the parser to stop are reported. Parsing stops after the first error.

The YACC program that is produced by MPLRUN is named microasm.y. This program is submitted to YACC, and YACC presents a listing of the attempt to compile it. The normal course for this listing is:

```
yacc microasm.y

conflicts: <integer> shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.c microasm.o
```

Abnormalities may occur if the file microasm.y does not meet YACC's expectations. As far as AMISS is concerned, the most likely abnormalities are:

1) An abnormal course listing may be of the form:

```
fatal error
        non terminal <name> not defined
line number <number>
***** ERROR code 1
```

This will occur if in the language description file submitted to MPLRUN, a partname that does not exist (ie. was not named in the @ PARTS section of the microarchitecture) is used to define a grammar rule. For example, no part named ACC is listed in the HDL program but a grammar rule:

```
| ACC "=" R0  {OPCODE = 0;
               ACNTRL = 1;
              }
```

is listed.

It will also occur if a word that is not a valid MPL keyword is used. For example, a mistype of "BITS" instead of BIT in:

```
| IF BITS "(" one "," two ")"
```

2) If a control field is mistyped in the actions, for example, OPSCODE instead of OPCODE, a listing of the form:

```
cc -c  y.tab.c  "microasm.y",  line
<integer>: opscode undefined
```

will result.

3) If one rule is defined more than once, the following will be reported:

```
1 rule never reduced
conflicts:
        <integer> shift/reduce
        <integer> reduce/reduce
```

It is probable that if the rules are identical, the microassembler will still work. It is best to rmove the ambiguous rule.

BIBLIOGRAPHY

---

[1] ACM Curriculum Committee on Computer Science, "Curriculum Computer Science", CACM, Vol. 22, No.3, March, 1979.

[2] IEEE Education Committee, "A Curriculum in Computer Science and Engineering", preliminary version. Pub. EH0119-8 Jan. 1977.

[3] Baer, J.L., Computer Systems Architecture, Computer Press, 1980.

[4] Mano, M., Computer Systems Architecture , Prentice Hall, 1976.

[5] Tomek, I., Introduction to Computer Organization , Computer Science Press.

[6] Tannenbaum, A., Structured Computer Organization , Prentice Hall, 1983.

[7] Bic, L. MICOS: A Microprogrammable Computer Simulator , Computer Science Press, 1984.

[8] Parker, J., "A microprogramming simulator for instructional use", ACM SIGSCE BULL, vol. 16, no 1, p 69-75, Feb 1984.

[9] Smith, M., "A microprogrammable microprocessor simulator and development system", IEEE Trans. on Education , vol E-27, No. 2, p 93-100, May 1984.

[10] Bell Laboratories, Inc., UNIX Programmers Manual , Volume 2.", New York: Holt, Rinehart and Winston , 1983, pp 351-400.

[11] Kernighan, B., Pike, R., The UNIX Programming Environment , Englewood Cliffs, N.J.: Prentice-Hall, 1984.

[12] Kelly, A., Pohl, I., A book on C , Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1984.

[13] Kernighan, B., Ritchie, D., The C Programming Language , Englewood Cliffs, N.J.: Prentice-Hall.