HOPKINS COMPUTER RESEARCH REPORTS

REPORT #31

MARCH 1974

THE JHU UNIVERSAL HOST MACHINE II - PART I: THE MACHINE

BY

J. W. DAVISON

RESEARCH PROGRAM IN COMPUTER SYSTEMS ARCHITECTURE

COMPUTER SCIENCE PROGRAM

THE JOHNS HOPKINS UNIVERSITY

BALTIMORE, MARYLAND

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

HOPKINS COMPUTER RESEARCH REPORT

#31

The JHU Universal Host Machine II

Part I: The Machine*

J. W. Davison

*This work was supported, in part, by the U. S. Atomic Energy Commission under Contract AT (11-1 3288)

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

CONTENTS

PART I The Machine

1. Introduction
2. The Simulator
 - 2.1 General
 - 2.2 Simulator Organization
 - 2.2.1 Overall Functional Organization
 - 2.2.2 Program Organization
3. The Microassembler and Language Format
 - 3.1 General
 - 3.2 Microassembler Organization
 - 3.2.1 Overall Functional Organization
 - 3.2.2 Program Organization
 - 3.3 The Microassembly Language Format
4. The Machine: JHUHM II
 - 4.1 Basic Organization
 - 4.1.1 General
 - 4.1.2 Memory Resources
 - 4.1.3 Transformational Resources
 - 4.2 Instruction Set
 - 4.2.1 Machine Instructions
 - 4.2.2 Simulator Instructions
5. The Microassembly Language: MICRO II
 - 5.1 General
 - 5.2 Pseudo Instructions
 - 5.3 Instructions
 - 5.4 Assembler Output
6. Evaluation and Comments
7. Listings (Not included; available on request)

PART II An Emulation and Evaluation (Separately published;
Author: S. Bradley)

1. The Emulator

2. Evaluation

1. INTRODUCTION

By "Universal Host Machine" is meant a machine on which it would be possible to "effectively" emulate (almost) any computer. The development of dynamically microprogrammable computers appears to offer great promise in this respect.

In order to investigate this area, a microprogramming seminar was held in the fall of 1970. During this seminar, a simulator and assembler for a microprogrammed host machine were developed. The result of this effort was reported in [1].

In the Fall of 1972, a similar seminar was conducted. During this seminar, the simulator and assembler were modified to model a new host machine. This new machine was designed to exhibit greater parallelism than the previous one. In addition, realizing that this would not be the last such revision, considerable effort was expended in making the simulator and assembler easily adapted to new models. A conscious effort was also made to keep the programs as independent of the implementing machine as possible.

Part I of this report contains a description of the revised simulator and microassembler, as well as a description and partial evaluation of the new micromachine. Part II (separately published) describes an emulation of part of the IBM 360 and further evaluates the machine.

2. THE SIMULATOR

2.1 GENERAL

The current simulator is a generalization of the previous version. It is designed to be more easily adapted to new machines. The simulator is written in FORTRAN IV, for the most part. The logical and twos-complement arithmetic routines are the only portions written in assembly language. Hopefully, the simulator will be easily transported to other computers.

Perhaps the most significant generalization of the simulator has been the freeing of the simulated machine from a strict dependence on the word-length of the base machine. The simulator will now model any machine with a word the same size or smaller than that of the base machine (currently 36 bits on a 7094). The advantage here is that if, as with JHUHM II, the simulated machine has a shorter wordlength than the base machine, the microprogrammer may ignore completely the base machine and program only the simulated machine. Otherwise, the programmer must be aware of the length of words in the base machine, especially when shifting or rotating data.

Running under IBJOB on an IBM 7094, the program simulates the current machine at a rate of approximately 800 microinstructions per second. This performance is obtained by rewriting the critical routines GETBIT, SINK, and SOURCE (discussed below) in assembly language. If these routines remain in FORTRAN, the rate is approximately 400 microinstructions per second.

2.2 SIMULATOR ORGANIZATION

2.2.1 Overall Functional Organization

The simulator is organized essentially as the control portion of a generalized machine. It cycles through the following sequence:

- a) fetch next instruction
- b) use first five bits of instruction to select instruction processor
- c) update clock, memory and instruction counter
- d) loop to a)

This sequence is built into the main driving routine, SIMULA. The first instruction is taken from location 0 of the control store.

A memory hierarchy is also built into the simulator. Presently, there are two banks of 16 registers each, 1024 words of control store, and a 16384 word main memory.

A set of utility routines is provided to allow the user easily to program the instruction processors. Included among these routines are facilities to truncate any word to the simulated machine word length, to extract bits from the Micro Instruction Register (MIR), to store or retrieve an item in any one of the memories, and to perform twos complement addition. All of these routines work in the simulated machine wordsize. There are also routines to perform logical and shifting operations. These work in the base machine, but the truncation routine mentioned above may be used to truncate the results. If the simulated word is smaller than the base machine's, the simulated word is right justified in the base machine.

In retrospect, after using the simulator, a slightly different organization might have been more appropriate. This would be the selection of one of four (or eight) instruction decoders on the basis of the first few bits of the instruction, rather than the selection of instruction processors directly. This might result in a slightly smaller system, and would conform more closely to typical machine organization.

2.2.2 Program Organization

The program consists of a mainline routine which initializes the

simulator and loads the memory. It then calls SIMULA, the routine which drives the simulation. The mainline routine loads the memories by reading the data from the standard input file. Control information specifies which memory is to be loaded, and when to start the simulation. Control cards are signalled by an "L" or "G" in column 1. An "L" indicates the cards which follow are to be loaded into the memory shown in column 6. Each data card specifies a location and its content. A "G" in column 1 indicates completion of loading, and starts the simulation.

SIMULA, as mentioned in the last section, fetches the next instruction into the MIR. After extracting the first 5 bits from the MIR, if they are not all zero, the appropriate instruction routine is called. If the first 5 bits are all zero, the instruction may refer to one of the simulator directives, so the first 8 bits are checked. If they are all zero, the instruction is assumed to be a machine NOOP. (note however, that if more than one operation is specified in a single instruction, more processing may be required.) If bits 5-7 are not all zero, the instruction is a simulator directive and is interpreted accordingly. The simulator directives and their effect is shown in table 2.1. The syntax of these directives will be discussed in section 4.2.2, and the formats are shown in Figure 4.

The following subroutines/functions are available for use in constructing instruction processors:

Name	Args	Type	Function
GETBIT	Lo,H1	FUNCTION	Extracts bits Lo through H1 from MIR
SOURCE	Memory,Locn	FUNCTION	Fetches contents of Memory(locn)
SINK	Memory,Locn, Contnt	SUBROUTINE	Stores Contnt in Memory(Locn)
ADDER	Op1,Op2, Carry	SUBROUTINE	Two's complement add of Op1, Op2, and Carry, sets C,V condition codes
HWIFE	Time	SUBROUTINE	Updates CLOCK by Time, sets MIC to new value, and clocks main memory operations on current machine
NOSUCH		SUBROUTINE	Prints MIR, MIC, and CLOCK, and "UNDEFINED INSTRUCTION"
JAND	Op1,Op2	FUNCTION	Logical AND
JIOR	Op1,Op2	FUNCTION	Logical Inclusive OR
JEOR	Op1,Op2	FUNCTION	Logical Exclusive OR
JNOT	Op1	FUNCTION	Logical Complement
JLS	Op1,Amount	FUNCTION	Logical Left Shift
JRS	Op1,Amount	FUNCTION	Logical Right Shift
JRL	Op1,Amount	FUNCTION	Rotate Left
CHOP	Op1	FUNCTION	Truncates to SIMLEN

TABLE 2.1 SIMULATOR DIRECTIVES

Opcode		Function
Symbolic	Numeric	
STOP	1	Exits from simulator, end of program
DUMP	2	Prints value of CLOCK, and all registers
PRR	3	Prints up to 4 selected registers
PRM	4	Prints a block of up to 128 words from SR2 or SR3
PRC	5	Prints value of CLOCK, CCODES, and next value of MIC, MIR

NOTE: None of these simulator directives effect the value of CLOCK.

3. THE MICROASSEMBLER AND LANGUAGE FORMAT

3.1 GENERAL

In order that the simulator be useful, some form of programming language for the simulated machine is obviously needed. This language must serve dual purposes: easing the programming of the simulated machine, and facilitating the evaluation of the simulated machine design. The second of these purposes, design evaluation, implies that the language must closely reflect the machine structure. A more or less traditional assembly language seems suitable. The fact that the simulator is intended to be used for examining a variety of machines imposes a complication; each of these machines will, necessarily, have a different language. Further, each language can be expected to change as the design matures. For these reasons, the translator should be general enough to handle a wide variety of languages.

These considerations, plus the existence of a microassembler for the previous version of the simulator, led to the current microassembler. As it now stands, the general format of the language is fixed by the assembler, but specific opcodes and machine format codes are specified in a table. This table is easily changed to modify the language as a given machine design changes. In order to adapt the translator to a new machine, it may be necessary also to change one or more of the code generation routines. These changes should not be difficult, however, since they consist mostly of a sequence of calls to standardized utility and parsing routines.

3.2 ASSEMBLER ORGANIZATION

3.2.1 Overall Functional Organization

The microassembler is of the two pass variety. During the first pass, a symbol table is generated; all other processing is left to the second pass. The basic functioning of the second pass is as follows:

Card images are read in and fully processed, one at a time.

Each comment card is ignored, except for listing.

The processing of each non-comment statement begins by obtaining the symbolic opcode. This opcode is used to access a table for:

- a) the numeric opcode
- b) the format code for the instruction
- c) extended opcode information, if needed.

The format code is used to select one of several code generators which process the remainder of the statement.

After the code has been generated, the statement and the machine code is written to the appropriate output files.

Most of the processing in the assembler is done by a set of standardized subroutines which will be described in more detail in section 3.2.2.

As can be seen from the above description, the addition or deletion of instructions from a language may, generally, be accomplished merely by updating the table. By changing the code generation routines, a new machine instruction format can be implemented. It is only when changes in the basic format of the language (such as the number of characters allowed in a symbol, or changes in card format) are required that modification of other parts of the assembler will be necessary. Even such changes will usually be restricted to two or three subroutines.

3.2.2 Program Organization

The assembler consists of a mainline routine which handles initialization and then calls PASS1 and PASS2, in that order. On completion of PASS2, the current system terminates. This routine can be readily modified to call the simulator after a successful completion--assuming, of course, that both the assembler and the simulator can be in core at the same time.

PASS1, as its name implies, is the first pass processor. It constructs the symbol table, which is hash coded. PASS1 is concerned only with labels and with the pseudo-operations EQU, BSS, and END. In order to make PASS1 as simple as possible, symbols may have values assigned in only the following ways:

- a). Symbol is a label for any instruction other than EQU.
In this case, the value of the symbol is the current value of the location counter.
- b). Symbol is a label for an EQU instruction.
In this case, the value of the symbol is the value of the expression which is the only operand. In the current implementation, the expression must be a single constant.

PASS1 makes use of two standardized routines: NUM and ERROR.

PASS2 is the second pass processor. As previously described, it uses the symbolic opcode to access a table for information on how to continue processing. The table is organized so that it can be searched using a binary search technique. The code generators, which are selected using information from the table, consist of a sequence of calls to subroutine NXTFLD, followed by a sequence of uses of functions JLS and, possibly, JIOR. NXTFLD is used to parse a segment of the statement, and the data so obtained is combined using JLS and JIOR to generate a machine language instruction.

NXTFLD is the basic parsing routine for the assembler. This routine determines most of the basic format for the assembly language. This format consists primarily of a hierarchy of separators/terminators. The hierarchy is as follows:

- a) a comma or a single blank (note that multiple blanks signal an error)
- b) a slash
- c) an asterisk

Actually, so far as NXTFLD is concerned, there is no difference between the slash and the asterisk; the difference between them is maintained by the code generators in PASS2. NXTFLD shares a current character pointer, INDEX, with the code generators, and when called it treats the string between INDEX and the next separator as a token. If the token is null, (i.e. INDEX points to a separator) or an error is encountered, NXTFLD returns zero as the value of the token; otherwise, if the token is a symbol, its value is obtained from the symbol table. If the token is a literal or a constant, NUM is called to evaluate it. The difference between the separators is only that INDEX is advanced to point to the next character if a comma or a blank is encountered, but is left pointing to the separator if it is a slash or an asterisk. The code generators move INDEX beyond a slash whenever they complete processing a segment of code, and INDEX does not move past an asterisk. The effect of these machinations is to allow the omission of null field markers at the right end of each segment, and to allow the omission of null segments at the end of a statement. Anything following an asterisk is thus treated as a comment. The primary reason for this hierarchy is to simplify the specification of parallel operations in a single statement.

NUM is used to convert decimal, octal, or hexadecimal constants into integers.

Table 3.1 Other routines and their function.

Name	Used By	Function
ERROR	PASS1,PASS2,NXTFLD,NUM	Prints error messages.
LIST	PASS2	Handles normal output
CHOP	PASS2	Truncates constants to word length of simulated machine.
GETCHR	NXTFLD,NUM	Gets next character from input.
APPEND	NXTFLD	Appends character to token
JIOR	PASS2,NXTFLD,NUM	Logical Inclusive OR
JAND	PASS2,NXTFLD	Logical AND
JLS	PASS2,NUM	Logical Left Shift
JRL	PASS2,NXTFLD	Logical Rotate Left

3.3 THE MICROASSEMBLY LANGUAGE FORMAT

As was noted in previous sections, the overall format of any micro-assembly language which will be handled by this translator is fixed, although the particular format of individual languages may vary somewhat.

In addition to the separator hierarchy described in the description of NXTFLD in section 3.2.2, there is a particular card format required, and symbolic names are restricted to no more than five characters. Table 3.2 is a BNF description of the general format for the languages.

The assembler also recognizes one control card. This card must be the first card in the deck, and must have "*DECK*" in columns 1-6. If such a card is included, the assembler will produce an object language card deck suitable for input to the simulator.

The required card format is:

1	8	16	<-- Card columns
---	---	----	------------------

Comment cards:

*<comment>

Other cards:

<label> <opcode> <operands><eol><comment>

TABLE 3.2 Language Format

<letter>	::=	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
<digit>	::=	0 1 2 3 4 5 6 7 8 9
<oct it>	::=	0 1 2 3 4 5 6 7
<hex it>	::=	0 1 2 3 4 5 6 7 8 9 A B C D E F
<alpha>	::=	<letter> <digit> <null>
<blank>	::=	" "
<character>	::=	<alpha> ' \$ () <blank> '"=" ~ + * . , /
<null>	::=	
<dec const>	::=	<digit> <digit><dec const>
<oct const>	::=	0<oct string>
<hex const>	::=	H<hex string>
<oct string>	::=	<oct it> <oct it><oct string>
<hex string>	::=	<hex it> <hex it><hex string>
<constant>	::=	<dec const> <oct const> <hex const>
<literal>	::=	'='<constant>
<name>	::=	<letter><alpha><alpha><alpha><alpha>
<label>	::=	<name> <null>
<token>	::=	<name> <literal> <dec const>
<sep a>	::=	, <blank>
<sep b>	::=	/
<sep c>	::=	*
<eol>	::=	<sep a> <sep b> <sep c>
<opcode>	::=	<letter><alpha><alpha><alpha>
<operands>	::=	<segment> <segment><sep b><operands>
<segment>	::=	<field> <field><sep a><segment>
<field>	::=	<token> <null>
<comment>	::=	<character> <character><comment>

4. The Machine: JHUHM II

4.1 ORGANIZATION

4.1.1 General

JHUHM II, the Johns Hopkins Universal Host Machine, attempt II, is a 32-bit two's complement machine with a limited capability for parallel or overlapped use of internal resources. Figure 1 is a conceptual view of the general organization and major data paths in the machine. The switches shown are intended to convey that there is a selection between the paths shown, and that only one may be used at a time. All other paths are available concurrently during the execution of a single instruction. Thus, during a typical instruction, an ALU operation, a memory operation, and a register to register transfer may occur.

4.1.2 Memory Resources

JHUHM II has four storage resources, designated SR0, SR1, SR2, and SR3. SR0 and SR1 are fast access register banks of 16 registers each. SR2 is a 1Kword high speed memory. SR3 is a 16Kword normal/medium speed memory.

SR0 is considered as dual purpose. In general, each of these registers is intended to be used either as a general purpose register or as the source or sink in a memory operation. However, two of the registers are used as special purpose registers. Register 0 is hardwired to zero, and register 1 is the Micro Instruction Register, MIR.

SR1 is also dual purpose. Each register may be used as a general purpose register, or to hold the address in a memory operation. Again, some of the registers are special purpose. Register 1 is the Micro Instruction Counter, MIC, and registers 14 and 15 are used as accumulators, AC0 and AC1, respectively. The result of any ALU/shift operation is directed to either AC0 or AC1, as specified in the instruction.

SR2 is used as the control memory, or micromemory, and its cycle time determines the instruction time. Thus, instruction time is approximately 2 cycles allowing both I-fetch and a micromemory operation in one instruction time.

SR3 is the main memory. It operates at approximately one tenth the speed of SR2. It is assumed to operate concurrently with all other processing, essentially as an independent device. Thus, any instruction which may conduct a memory operation may initiate a main memory operation. This operation will be complete by the beginning of the fifth following instruction. If an attempt is made to initiate another main memory operation before this one completes, all other processing is delayed until the first main memory operation completes.

The operation of SR3 is assumed not to interfere with the operation of SR2 except during the initiation of the operation. Thus overlapped access to SR2 is explicitly possible during the operation of SR3.

In addition to these storage resources, there are two special purpose registers not directly accessible to the programmer. These are the clock and the status register. The clock counts the number of instruction cycles since beginning execution. It is of interest only for the evaluation of a machine or program during simulation. The simulator displays the clock value whenever the DUMP or PRC simulator directives are executed. The status register contains the condition codes. The meaning and how or when they are changed is shown in figure 2.

4.1.3 Transformational Resources

The transformational resources available in JHUHM II are an Arithmetic/Logical Unit and a shifter. The ALU is capable of the operations AND, OR (inclusive), XOR (exclusive OR), NOT (logical complement), COM (two's complement), and two's complement addition. The shifter can perform shifts or

rotates, up to 32 bits, either left or right, in either logical or arithmetic mode.

4.2 INSTRUCTION SET

4.2.1 Machine Instructions

There are two main types of instructions in JHUHM II, control instructions and transformational instructions. The control instructions can specify a conditional branch and a register to register transfer. The transformational instructions specify an ALU/shifter operation, a memory operation, and a register to register transfer.

The instructions are divided into four subgroups, with formats as shown in figure 3. In the one register ALU instructions, the Amount field is used only with the Increment instruction. Figure 5 is a table of opcodes. The opcode shown is the FMT field concatenated with the OP field. Table 4.2 is a concise description of each instruction. The notation used should not be difficult to understand, and is briefly described in Table 4.1.

The following is a more detailed discussion of some of the instructions. The Branch instructions may be either direct, or indirect. Bit 5 in the instruction is '1' for indirect branching. On a direct branch, the next instruction to be executed comes from the address specified in the instruction, while on an indirect branch, the address field of the instruction specifies a register in SR1. The address of the next instruction is taken from that register. The Mask field is used to designate which condition codes determine whether or not the branch occurs. Only those codes marked by 1's in the mask are examined. The four possibilities which are tested, and the instruction which branches on each condition are as follows:

<u>Instruction</u>	<u>Condition of tested bits</u>
BCA -- Branch on Condition, All	All set to 1
BNA -- Branch on Not condition, All	At least one is 0
BCY -- Branch on Condition, any	At least one is 1
BNY -- Branch on Not condition, any	All set to 0

An unconditional branch can be forced by using a mask of all 0's in the BCA instruction. To determine the effect of other special cases, see the definition in Table 4.2.

In the Increment instruction, INC, the Amount field has the high bit extended to make a full word prior to being sent to the adder. Thus, this allows incrementing a word by a value in the range -16 to +15.

On a memory operation, only one register is specified. The interpretation is that the address involved is in the specified register in SR1, and the data is, or should be placed, in the corresponding register in SR0.

Table 4.1 Notation

<u>Occurance</u>	<u>Meaning</u>
+	Addition
-	Two's complement
~	Logical complement
&	Logical AND
	Logical OR (inclusive)
\$	Logical OR (exclusive)
<	Left shift
>	Right shift
=>	Assignment to symbol on the right
(0,3)	An address: location 3 in SR0
(SI,R1)	An address: the location specified by the R1 field of the instruction in the memory specified in the SI field. In general, in any expression occurring in such an address, any variable refers to the content of the corresponding instruction field.
c(2,17)	The Contents of location 17 of SR2.
(0,7)[6]	Bit 6 of (0,7)
(3,4)[0:3]	Bits 0(the high bit) through bit 3 of (3,4)
&/CCODES	CCODES[0] & CCODES[1] & CCODES[2] & ...
!/{MASK & ~CCODES}	{MASK[0] & ~CCODES[0]} ! {MASK[1] & ~CCOD ... Note: The / operator is taken from APL, and extends to the appropriate number of components. In the case of MASK and CCODES, there are only 6 bits involved.

TABLE 4.2 Instruction Semantics

<u>Mnemonic_Opcode</u>		<u>Semantics</u>
NOOP	00	;Memory;Gate
INC	01	$c(S1, R1) + \text{Amount} \Rightarrow \text{Ac}'; \text{SetVC}; \text{Memory}; \text{Gate}$
SET	02	$c(S1, R1)[0] \Rightarrow H$ { Remark: the condition code}; $c(S1, R1)[31] \Rightarrow L; 1/c(S1, R1)[1:30] \Rightarrow Z;$ Memory;Gate
NOT	04	$\sim c(S1, R1) \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
COM	05	$-c(S1, R1) \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
ADD	10	$c(S1, R1) + c(S2, R2) \Rightarrow \text{Ac}'; \text{SetVC}; \text{Memory}; \text{Gate}$
ADC	11	$c(S1, R1) + c(S2, R2) + C \Rightarrow \text{Ac}'; \text{SetVC}; \text{Memory}; \text{Gate}$
AND	14	$c(S1, R1) \& c(S2, R2) \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
OR	16	$c(S1, R1) c(S2, R2) \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
XOR	17	$c(S1, R1) \oplus c(S2, R2) \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
SAL	20	$c(S1, R1) \ll \text{Amount} \Rightarrow \text{Ac}'; \text{SetVC}; \text{Memory}; \text{Gate}$
SAR	21	$c(S1, R1) \gg \text{Amount}$ {fill with $c(S1, R1)[0]$ } $\Rightarrow \text{Ac}';$ Memory;Gate
SLL	22	$c(S1, R1) \ll \text{Amount} \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
SLR	23	$c(S1, R1) \gg \text{Amount} \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
RLL	26	$c(S1, R1) \text{Rotate left Amount} \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
RLR	27	$c(S1, R1) \text{Rotate right Amount} \Rightarrow \text{Ac}'; \text{Memory}; \text{Gate}$
BCA	30	$\text{Address}' \Rightarrow \text{MIC}$ if $1/\{\text{MASK} \& \sim \text{CCODES}\} = 0; \text{Gate}$
BNA	31	$\text{Address}' \Rightarrow \text{MIC}$ if $1/\{\text{MASK} \& \sim \text{CCODES}\} = 1; \text{Gate}$
BCY	32	$\text{Address}' \Rightarrow \text{MIC}$ if $1/\{\text{MASK} \& \text{CCODES}\} = 1; \text{Gate}$
BNY	33	$\text{Address}' \Rightarrow \text{MIC}$ if $1/\{\text{MASK} \& \text{CCODES}\} = 0; \text{Gate}$
SetVC		Carry out of Result[0] $\Rightarrow C;$ Carry into Result[0] $\Rightarrow V$
Memory		If D=0 Then If R3=0 Then NOP Else $c(2+M, c(1, R3)) \Rightarrow (0, R3);$ If D=1 Then $c(0, R3) \Rightarrow (2+M, c(1, R3))$
Gate		If $(S5, R5) = (0, 0)$ Then NOP Else $c(S4, R4) \Rightarrow (S5, R5)$
Ac'		$(1, \text{Ac} + 14)$
Amount'		Amount with high order bit extended

Address	Address if direct, c(0,Address) if indirect
H,Z,L,C,V,B	CCODES[0],[1],[2],[3],[4],[5] respectively
MIC	(1,1)

4.2.2 Simulator Directives.

In addition to the machine instructions described above, there are five pseudo-instructions which are actually simulator directives. These directives provide a limited output capability, to enable the monitoring of the micromachine. These directives were briefly described in section 2.2.1, Table 2.1. Figure 4 shows the formats of these directives. STOP, DUMP, and PRC have only an opcode field. The remainder of the word is ignored. The PRR directive (Print Registers) has four additional subfields, each of five bits, which specify four registers to be displayed. In the PRM directive (Print Memory), bit 8, M, specifies either SR2 (M=0), or SR3. Bits 9-24 give a starting address, and bits 25-31 specify the number of words to be displayed. The contents of all registers and memory words, as well as identifying information, dumped in this manner are given on octal unless otherwise noted when printed out.

5. The Microassembly Language: MICRO II

5.1 GENERAL

The basic format of the language is similar to that outlined in section 3. As explained in section 3.2, code generators in the assembler are selected on the basis of a format code in the opcode table. Although the assembler provides for eight format divisions, MICRO II makes use of only five. These five are:

- 1). Assembler Directives
- 2). Double Operand with Accumulator
- 3). Single Operand with Accumulator
- 4). Single Operand without Accumulator
- 5). Branch Instructions

In MICRO II, the registers are considered as one bank of 32 rather than two banks of 16. SR0 consists of the registers 0 through 16, and SR1 consists of the registers 16 through 31. These may be assigned any mnemonic desired through the use of the EQU directive.

5.2 PSEUDO INSTRUCTIONS

There are two types of pseudo instructions: Assembler directives, and Simulator directives. Six assembler directives are provided to permit some flexibility in programming style. These directives, their formats and effects are described below. The metasymbols are as defined in table 3.2. The symbol 'LC' refers to the current value of the location counter. LC is initially set to 0. 'c(LC)' means the contents of SR2(LC). 'v(xx)' means the value of xx, where the value is that which would be returned by the appropriate routine, NUM or NXTFLD. These routines are briefly described in section 3.2.2.

SYNTAX			SEMANTICS
1	8	16	<-- Card Columns
<label>	BSS	<constant>	PASS1: v(label):=LC;next LC:=LC+<constant> PASS2: LC:=LC+<constant>
<label>	CON	<token>	PASS1: v(label):=LC;next LC:=LC+1 PASS2: c(LC):=v(token);next LC:=LC+1
	END		PASS1: terminates Pass 1 PASS2: terminates Pass 2
<name>	EQU	<constant>	PASS1: v(label):=<constant> PASS2: ignored
<label>	HEX	<hex string>	PASS1: v(label):=LC;next LC:=LC+1 PASS2: c(LC):=v(hex string);next LC:=LC+1
<label>	OCT	<oct string>	PASS1: v(label):=LC;next LC:=LC+1 PASS2: c(LC):=v(oct string);next LC:=LC+1

The semantics of simulator directives has already been discussed. The syntax of the assembly language versions is shown below. The maximum number of fields is shown. As with all other assembly language statements, fields

may be omitted from the right end of a segment if the segment terminates with either a '/' or a '*'.

```

1      8      16      <-- Card Columns

<label>  <opx>                                <opx> is STOP,DUMP, or PRC

<label>  PRR   <reg1>,<reg2>,<reg3>,<reg4>

<label>  PRM   <memory>,<location>,<Length>

```

In PASS2, pseudo instructions are handled as FMT code 1 instructions, with extended opcode information provided to select the particular code generator for that instruction.

5.3 INSTRUCTIONS

The semantics of the various machine instructions was discussed in section 4.2. This section will treat the syntax of the instructions. The instructions will be treated by code generator formats. In the following schemas the metasymbols <memop> and <gateop> frequently appear; they are defined as follows:

<memop> ::= <memory>,<read/write>,<register>

<gateop> ::= <source reg>,<sink reg>

It should also be noted that in these and the other definitions the values of the various fields must be in the ranges indicated:

Field	Range	Remarks
<accum>,<memory>, <read/write>	0,1	
<op1>,<op2>,<reg>, <register>, <source reg>, <sink reg>	0 through 31	If required by field size in the generated code, high order bits will be truncated.
<address>	0 through 1023	
<mask>	0 through 63	Six bits

TABLE 5.1 Syntax of Symbolic Instructions by FMT Class.

1 8 16 <----- Card Columns

FMT=1: extended_opcode = 0

<label> NOOP <null>/<memop>/<gateop><eol>

FMT=2: Double_operand_instructions_with_AC

<label> <op> <op1>,<op2>,<accum>/<memop>/<gateop><eol>

 <op> ::= ADC|ADD|AND|INC|OR|XOR|RL|RLR|SAL|SAR|SLL|SLR

FMT=3: Single_operand_instructions_with_AC

<label> <op> <op1>,<accum>/<memop>/<gateop><eol>

 <op> ::= COM|NOT

FMT=4: Single_operand_instruction_without_AC

<label> SET <reg>/<memop>/<gateop><eol>

FMT=5: Branch_instructions

<label> <op> <mask>,<address>/<gateop><eol>

 <op> ::= BCA|BCA*|BCY|BCY*|BNA|BNA*|BNY|BNY*
 (the * indicates indirect addressing--bit 5 is 1)

5.4 ASSEMBLER OUTPUT

Beginning on the following page is an example of the output of the microassembler. This program is a partial emulation of the IBM 360 and will be discussed in Part II of this report.

PASS1

HASHING CONFLICT	R11	MICRO	449
HASHING CONFLICT	R23	UNCON	262
HASHING CONFLICT	HEX11	VCON	463
HASHING CONFLICT	TST0	R18	10
HASHING CONFLICT	NR	ACC1	166
HASHING CONFLICT	NR	R28	164
HASHING CONFLICT	LA	HEX10	399
HASHING CONFLICT	CHECK	MIC	84
HASHING CONFLICT	LNR	LA	397

DECK

360 EMULATOR WRITTEN IN UHM MICROCO

```

*
*
*
*
*
UNCON EQU 0 UNCONDITIONAL BRANCH INDICATOR
MICRO EQU 0 MICRO MEMORY FLAG
*
REGISTER EQUATES
R0 EQU 0 (0,0) - ALWAYS CONTAINS 0
R1 EQU 1 (0,1) - MICRO INSTRUCTION REGIS
R2 EQU 2 (0,2)
R3 EQU 3 (0,3)
R4 EQU 4 (0,4)
R5 EQU 5 (0,5)
R6 EQU 6 (0,6)
R7 EQU 7 (0,7)
R8 EQU 8 (0,8)
R9 EQU 9 (0,9) - CONDITION CODE
R10 EQU 10 (0,10) - INSTRUCTION REGISTER
R11 EQU 11 (0,11)
R12 EQU 12 (0,12)
R13 EQU 13 (0,13)
R14 EQU 14 (0,14)
R15 EQU 15 (0,15)
R16 EQU 16 (1,0)
R17 EQU 17 (1,1) - MICRO INSTRUCTION COUNT
R18 EQU 18 (1,2)
R19 EQU 19 (1,3)
R20 EQU 20 (1,4)
R21 EQU 21 (1,5)
R22 EQU 22 (1,6)
R23 EQU 23 (1,7)
R24 EQU 24 (1,8)
R25 EQU 25 (1,9) -
R26 EQU 26 (1,10) - INSTRUCTION COUNTER
R27 EQU 27 (1,11)
R28 EQU 28 (1,12)
R29 EQU 29 (1,13)
R30 EQU 30 (1,14) - ACCUMULATOR 1
R31 EQU 31 (1,15) - ACCUMULATOR 2
*
SYMBOLIC FIELD EQUATES
MIR EQU 1 MICRO INSTRUCTION REGISTER
MIC EQU 17 MICRO INSTRUCTION COUNTER
IR EQU 10 INSTRUCTION REGISTER
IC EQU 26 INSTRUCTION COUNTER
MAIN EQU 1 MAIN MEMORY FLAG
WRITE EQU 1 WRITE FLAG FOR MEMORY OPERATION
READ EQU 0 READ FLAG FOR MEMORY OPERATIONS
AC1 EQU 0 FLAG BIT FOR ACCUMULATOR 1 = (1
AC2 EQU 1 FLAG BIT FOR ACCUMULATOR 2 = (1
ACC1 EQU 30 REF. TO ACCUMULATOR 1
ACC2 EQU 31 REF. TO ACCUMULATOR 2
CCODE EQU 9 CONDITION CODE
CON EQU 4 MASK FOR C BIT
VON EQU 2 MASK FOR V BIT
VCON EQU 6 MASK FOR V AND C BITS
LON EQU 8 MASK FOR L BIT
ZON EQU 16 MASK FOR Z BIT

```

		SIGN	EQU	32 MASK FOR H BIT
		TESTO	EQU	56 MASK FOR O TEST
		*		LOCATIONS 0 - 15 IN MICRO-MEMORY
		*		ARE THE 16 360 GENERAL PURPOSE REGI
0000	030000400020	GPRS	BCA	UNCON, START/R0, R16
			BSS	15
			BSS	16
0040	000100000000	EXIT	STOP	
0041	000200000000	PRINT	DUMP	
0042	000400000020		PRM	MICRO, 0, 16
0043	030000424000		BCA	UNCON, FETCH//
			BSS	1
		*		START OP CODE TABLE
0045	030000726000	HEX05	BCA	UNCON, BALR//
0046	030000730000	HEX06	BCA	UNCON, BCTR//
0047	030000646000	HEX07	BCA	UNCON, BCR//
			BSS	8
0060	030000732000	HEX10	BCA	UNCON, LPR//
0061	030000734000	HEX11	BCA	UNCON, LNR//
0062	030000736000	HEX12	BCA	UNCON, LTR//
0063	030000740000	HEX13	BCA	UNCON, LCR//
0064	030000534000	HEX14	BCA	UNCON, NR//
0065	030000742000	HEX15	BCA	UNCON, CLR//
0066	030000550000	HEX16	BCA	UNCON, OR//
0067	030000564000	HEX17	BCA	UNCON, XR//
0070	030000600000	HEX18	BCA	UNCON, LR//
0071	030000672000	HEX19	BCA	UNCON, CR//
0072	030000506000	HEX1A	BCA	UNCON, AR//
0073	030000606000	HEX1B	BCA	UNCON, SR//
0074	030000744000	HEX1C	BCA	UNCON, MR//
			BSS	1
0076	030000746000	HEX1E	BCA	UNCON, ALR//
0077	030000750000	HEX1F	BCA	UNCON, SLR//
			BSS	33
0141	030000620000	HEX41	BCA	UNCON, LA//
			BSS	4
0146	030000752000	HEX46	BCA	UNCON, BCT//
0147	030000754000	HEX47	BCA	UNCON, BC//
			BSS	8
0160	030000756000	HEX50	BCA	UNCON, ST//
			BSS	7
0170	030000760000	HEX58	BCA	UNCON, L//
			BSS	7
		*		END OP CODE TABLE
0200	030000405077	START	BCA	UNCON, INIT/MIC, ACC2
0201	000000000371	CBASE	CON	CAREA
0202	001760500032	INIT	INC	ACC2, 1, AC1/WRITE, MICRO, R16/R0, IC
0203	001000634000		INC	R0, 1, AC2/READ, MICRO, ACC1//
0204	001001001746		INC	R0, 2, AC1//ACC2, R6
0205	001001601707		INC	R0, 3, AC2//ACC1, R7
0206	001342401750		INC	R14, 5, AC1//ACC2, R8
0207	001343601725		INC	R14, 7, AC2//ACC1, R21
0210	001000001760		INC	R0, 0, AC1//ACC2, R16
0211	001000200731		INC	R0, 0, AC2//R14, R25 GATE THE ADDRESS
0212	001620064000	FETCH	INC	R25, 0, AC1/READ, MAIN, IC//
0213	001620634000		INC	R25, 1, AC2/READ, MICRO, ACC1//
0214	001621036702		INC	R25, 2, AC1/READ, MICRO, ACC2/R14, R2 CD
		*		READ IN FOUR MASKS FROM MICRO MEMO
0215	001621634753		INC	R25, 3, AC2/READ, MICRO, ACC1/R15, R11
0216	001622036714		INC	R25, 4, AC1/READ, MICRO, ACC2/R14, R12

0217	002240034755	SET	MAIN MEMORY FETCH NOW COMPLETED
0220	033340100733	BNY	IR/READ,MICRO,ACC1/R15,R13
0221	001640400000	INC	TEST0,EXIT/R14,R27
0222	014265201732	AND	IC,1,AC1///
0223	023774000000	SLR	R11,IR,AC2//ACC1,IC
0224	010775601722	ADD	ACC2,24,AC1///
0225	014305000000	AND	ACC2,R27,AC2//ACC1,R18
0226	023750000000	SLR	R12,IR,AC1///
0227	014325201724	AND	ACC1,16,AC1///
0230	023772200000	SLR	R13,IR,AC2//ACC1,R20
0231	030010470000	BCA	ACC2,20,AC2///
0232	010051011763	ADD	VON,RXRS//
0233	030400034000	BCA*	R2,R18,AC1/READ,MICRO,R20/ACC2,R19
0234	001623013763	INC	ACTUAL BRANCH ADDRESS IS IN ACC1 =
0235	014242634274	AND	UNCON,R14// BRANCH THROUGH OP COD
0236	023766201033	SLR	R25,6,AC1/READ,MICRO,R21/ACC2,R19
0237	014247027765	AND	IR,R5,AC2/READ,MICRO,ACC1/R5,R28
0240	000000000735	NOOP	ACC2,12,AC2//R16,R27
0241	010051211726	ADD	IR,R14,AC1/READ,MICRO,R27/ACC2,R21
0242	030400036000	BCA*	//R14,R29
0243	002000006000	SET	R2,R18,AC2/READ,MICRO,R20/ACC1,R22
0244	010062200011	ADD	ACTUAL BRANCH ADDRESS IS IN ACC2 =
0245	002760001743	SET	UNCON,R15// BRANCH THROUGH OP COD
0246	000000106000	NOOP	RO/READ,MICRO,R19//
0247	030010530000	BCA	R3,R4,AC2//RO,CCODE
0250	030020424411	BCA	ACC2//ACC2,R3
0251	033340424000	BNY	/WRITE,MICRO,R19//
0252	032200424351	BNA	VON,OVF//
0253	030000424311	BCA	CON,FETCH/R8,CCODE
0254	032020424411	BNA	TEST0,FETCH//
0255	030000522000	BCA	SIGN,FETCH/R7,CCODE
0256	002000006000	SET	UNCON,FETCH/R6,CCODE
0257	014062200011	AND	CON,FETCH/R8,CCODE
0260	002760001743	SET	UNCON,TST0//
0261	001000506000	INC	RO/READ,MICRO,R19//
0262	031340425711	BCY	R3,R4,AC2//RO,CCODE
0263	030000424011	BCA	ACC2//ACC2,R3
0264	002000006000	SET	RO,1,AC1/WRITE,MICRO,R19//
0265	016062200011	OR	TEST0,FETCH/ACC1,CCODE
0266	002760001743	SET	UNCON,FETCH/RO,CCODE
0267	001000506000	INC	RO/READ,MICRO,R19//
0270	031340425711	BCY	R3,R4,AC2//RO,CCODE
0271	030000424011	BCA	ACC2//ACC2,R3
0272	002000006000	SET	RO,1,AC1/WRITE,MICRO,R19//
0273	017062200011	XOR	TEST0,FETCH/ACC1,CCODE
0274	002760001743	SET	UNCON,FETCH/RO,CCODE
0275	001000506000	INC	RO/READ,MICRO,R19//
0276	031340425711	BCY	R3,R4,AC2//RO,CCODE
0277	030000424011	BCA	ACC2//ACC2,R3
0300	000000000203	NOOP	RO,1,AC1/WRITE,MICRO,R19//
0301	000000106000	NOOP	TEST0,FETCH/ACC1,CCODE
0302	030000424000	BCA	UNCON,FETCH/RO,CCODE
0303	004100006000	NOT	//R4,R3
0304	010741600011	ADD	/WRITE,MICRO,R19//
0305	001760400000	INC	UNCON,FETCH//
0306	002740001703	SET	R4,AC1/READ,MICRO,R19//
0307	030000514000	BCA	ACC1,R3,AC2//RO,CCODE
			ACC2,1,AC1///
			ACC1//ACC1,R3
			UNCON,STORE//

0310	002500012036	LA	SET	R20/READ,MICRO,R21/RO,ACC1	
0311	033340626000	X2	BNY	TEST0,B2//	
0312	010002000000		ADD	RO,R4,AC1///	
0313	002520000000	B2	SET	R21///	
0314	033340634000		BNY	TEST0,D2//	
0315	010742400000		ADD	ACC1,R5,AC1///	
0316	010753200000	D2	ADD	ACC1,R22,AC2///	
0317	014765400000		AND	ACC2,R11,AC1///	
0320	001622601703		INC	R25,5,AC2//ACC1,R3	RESTORE B2 MASK
0321	001000107765		INC	RO,0,AC1/WRITE,MICRO,R19/ACC2,R21/	
0322	030000424000		BCA	UNCON,FETCH//	
0323	001004000000	BCR	INC	RO,8,AC1///	
0324	002220000000		SET	CCODE///	
0325	032040656000		BNA	LON,HIGH//	
0326	023740400000		SLR	ACC1,1,AC1///	
0327	032100662000	HIGH	BNA	ZON,CHECK//	
0330	023741000000		SLR	ACC1,2,AC1///	
0331	014477200000	CHECK	AND	R19,ACC1,AC2///	
0332	002760000000		SET	ACC2///	
0333	033340424000		BNY	TEST0,FETCH//	
0334	030000424232		BCA	UNCON,FETCH/R4,IC	
0335	002100006000	CR	SET	R4/READ,MICRO,R19//	
0336	030200720011		BCA	SIGN,NEG2/RO,CCODE	
0337	002060000000		SET	R3///	
0340	030200424311		BCA	SIGN,FETCH/R6,CCODE	
0341	004100000000	SUB	NOT	R4,AC1///	
0342	010741600000		ADD	ACC1,R3,AC2///	
0343	001760400000		INC	ACC2,1,AC1///	
0344	002740000000		SET	ACC1///	
0345	033340424000	EQUAL	BNY	TEST0,FETCH//	
0346	032200424351	AHI	BNA	SIGN,FETCH/R7,CCODE	
0347	030000424311	ALOW	BCA	UNCON,FETCH/R6,CCODE	
0350	002060000000	NEG2	SET	R3///	
0351	032200424351		BNA	SIGN,FETCH/R7,CCODE	
0352	030000702000		BCA	UNCON,SUB//	
0353	030000424000	BALR	BCA	UNCON,FETCH//	
0354	030000424000	BCTR	BCA	UNCON,FETCH//	
0355	030000424000	LPR	BCA	UNCON,FETCH//	
0356	030000424000	LNR	BCA	UNCON,FETCH//	
0357	030000424000	LTR	BCA	UNCON,FETCH//	
0360	030000424000	LCR	BCA	UNCON,FETCH//	
0361	030000424000	CLR	BCA	UNCON,FETCH//	
0362	030000424000	MR	BCA	UNCON,FETCH//	
0363	030000424000	ALR	BCA	UNCON,FETCH//	
0364	030000424000	SLR	BCA	UNCON,FETCH//	
0365	030000424000	BCT	BCA	UNCON,FETCH//	
0366	030000424000	BC	BCA	UNCON,FETCH//	
0367	030000424000	ST	BCA	UNCON,FETCH//	
0370	030000424000	L	BCA	UNCON,FETCH//	
0371	000000000040	CAREA	CON	32	OP CODE TABLE DISPLAC
0372	037700000000		HEX	FF000000	OP CODE MASK (8 BITS)
0373	000003600000		HEX	000F0000	R2 (OR X2) FIELD MASK
0374	000074000000		HEX	00F00000	R1 FIELD MASK (4 BITS)
0375	030000000000	ADDM	HEX	C0000000	ADD MASK FOR RR-RX TC
0376	000000170000		HEX	0000F000	B2 FIELD MASK (4 BITS)
0377	000000007777		HEX	00000FFF	D2 FIELD MASK (12 BIT)
0400	000077777777		HEX	00FFFFFF	24 BIT ADDRESS MASK
			END		

6. Evaluation and Recommendations

...
 When she was good,
 She was very,very good,
 But when she was bad,
 She was horrid!

Anon

It is an interesting coincidence that JHUHM II proves to be as useful as it is. The ability to perform three operations in parallel seems to be quite useful at first glance. However, examination of the emulator shows that the capability is most often used here to overcome inherent problems in the design of the machine. That is, it is usually used as follows: compute an address known at assembly time, read from that address, move the data out of the way. Thus, the addition of an instruction to load a given register from a given memory location would not only be faster, but would reduce the load on the ALU.

The dual accumulator system is more difficult to judge. Its prime advantage is the reduction in the amount of bits needed in the instruction to specify the destination of an operation. This is important because it is difficult to specify multiple operations in only 32 bits. If an accumulator system is to be used, it is useful to have two accumulators, in order to avoid the bottleneck which would otherwise arise. However, other systems that might be suitable include the specification of only two actions per instruction--an ALU/Shift and a Memory/register transfer operation. This would allow the specification of a destination register in the instruction. Another approach would be to have some method of supplying immediate data. Then it would suffice to have one of the source registers serve as the destination since it would no longer be necessary to keep masks and constants in the registers.

In general, the requirement to address memory only through the registers, coupled with the nearly complete lack of immediate data instructions, makes any evaluation of the utility of the other features almost impossible. The overlapped operation of main memory is effective in JHUHM II, but, if masks didn't need to be loaded into the registers, it's not clear that it would still be useful. Were it possible to address memory directly, and to supply immediate data, it is doubtful that the current division of resources would be effective. Especially if the accumulators were eliminated, it is not likely that all three instruction parts could be used simultaneously. A more appropriate division then might be the ALU/Shifter-Memory transfer split mentioned above.

If, as is probable, the machine is to be used primarily for emulation, some sort of "gate under mask" facility would be useful. This would be a simplified form of the IXO instruction of Cook and Flynn [2]. With this instruction, it would be possible to extract a field from one register and transfer it to another register, perhaps rotated. Given an independent unit to perform this operation, a fruitful division of resources might then be ALU-Extraction unit-Memory transfer.

In any future revision or rewriting of the simulator, facilities should be added to collect data concerning instruction and resource utilization. In addition to information on overlapped resource utilization by the micro-machine, it should be possible to collect information about the emulation of another machine. This latter objective could be achieved through the addition of new simulator directives. Information which might be of interest in this area would include amount of time spent waiting on main memory, number of microinstructions used to emulate one higher level instruction, simulated clock time per higher level instruction, etc.

REFERENCES

- [1] Yelowitz, Misra, Converse, and Regis; A SIMULATOR FOR A MICROPROGRAMMED COMPUTER, IT'S MICROASSEMBLER, AND AN EMULATION; Hopkins Computer Research Reports, Report #1, APRIL 1971. The Johns Hopkins University Baltimore, Maryland.
- [2] Cook, R.W. and M.J. Flynn; System Design of a Dynamic Microprocessor; IEEE TC Vol. C-19, No. 3, pp213-222 March, 1970

FIG 1

JHUHM II

GENERAL ORGANIZATION AND DATA PATHS

FIG 2 CONDITION CODES

H	Z	L	C	V	B
---	---	---	---	---	---

<u>CODE</u>	<u>MEANING (when set)</u>	<u>HOW/WHEN CHANGED</u>
H	Hi order bit (bit 0) is 1	SET instruction
Z	Bits 1 $\rightarrow$ 30 NOT ALL 0	SET instruction
L	Lo order bit (bit 31) is 1	SET instruction
C	Carry out of bit 0	Arithmetic instructions (ADD, ADC, INC, SAL)
V	Carry into bit 0	Same as C
B	Main Memory Busy	Set when mem op on SR3 in process.

FIG 3 INSTRUCTION FORMATS

3A. MAJOR DIVISIONS

ONE-REGISTER ALU	00	ALU1	MEM	GATE
TWO-REGISTER ALU	01	ALU2	MEM	GATE
SHIFT	10	SHIFT	MEM	GATE
BRANCH	11	BRANCH		GATE

3B. SUB-DIVISIONS

ALU1	2	4	5	6	9	10	14	15
	OP	S ₁	R ₁		Amount		Ac	
ALU2	2	4	5	6	9	10	14	15
	OP	S ₁	R ₁	S ₂	R ₂		Ac	
SHIFT	2	4	5	6	9	10	14	15
	OP	S ₁	R ₁		Amount		Ac	
BRANCH	2	4	5	6	11	12		21
	OP	I	MASK		ADDRESS			
MEM	16	17	18					21
	D	M	R ₃					
GATE	23	24		26	27	28		31
	S ₄	R ₄		S ₅	R ₅			

$$S_i = 0 \Rightarrow SR0$$

$$S_i = 1 \Rightarrow SR1$$

R_i specifies which register within S_i

$$Ac = 0 \Rightarrow AC0$$

$$Ac = 1 \Rightarrow AC1$$

$$I = 0 \Rightarrow \text{Direct, else indirect}$$

$$D = 0 \Rightarrow \text{READ}$$

$$D = 1 \Rightarrow \text{WRITE}$$

$$M = 0 \Rightarrow SR2$$

$$M = 1 \Rightarrow SR3$$

FIG 4 SIMULATOR DIRECTIVE FORMATS

STOP, DUMP, PRC

PRR

PRM

FIG 5 OPCODE TABLE

		op							
		0	1	2	3	4	5	6	7
FMT		000	001	010	011	100	101	110	111
0	00	NOOP	INC	SET		NOT	COM		
1	01	ADD	ADC			AND		OR	XOR
2	10	SAL	SAR	SLL	SLR			RLL	RLR
3	11	BCA	BCY	BNA	BNY				