A Microcoded Machine Simulator and Microcode Assembler in a FORTH Environment

A. Cotterman, R. Dixon, R. Grewe, G. Simpson

Department of Computer Science
Wright State University

Abstract

A FORTH program which provides a design tool for systems which contain a microcoded component was implemented and used in a computer architecture laboratory. The declaration of standard components such as registers, ALUs, busses, memories, and the connections is required. A sequencer and timing signals are implicit in the implementation. The microcode is written in a FORTH-like language which can be executed directly as a simulation or interpreted to produce a fixed horizontal microcode bit pattern for generating ROMs.

The direct execution of the microcode commands (rather than producing bit patterns and interpreting those instructions) gives a simpler, faster implementation. Further, the designer may concentrate on developing the design at a block level without considering some of the implementation details (such as microcode fields) which might change several times during the design cycle. However, the design is close enough to the hardware to be readily translated. Finally, the fact that the same code used for the simulation may be used for assembly of the microcode instructions (after the field patterns have been specified) saves time and reduces errors.

1. Introduction

At the Wright State University computer architecture laboratory a microcoded machine simulator and microcode generator have been developed using FORTH. These tools have been used as "hands-on" instructional aids in graduate courses in computer architecture, and are also being used to aid the in-house development of new architectures in ongoing research efforts. The simulator provides basic block-level functional control and data-flow simulation for a machine architecture based on a microcoded implementation. The user specifies basic functional units such as registers, ALUs, memories, and busses and the major data path connections between these. Then microcode is written using a set of FORTH words which are executed to simulate the actions of corresponding microinstructions. Basic interactive debugging facilities allow the user to test and alter the hardware organization and the microcode implementation. The user can then present the same machine and microcode specifications, along with special mapping information, to the microcode generator to obtain actual horizontal microcode which can then be used to generate ROMs for a hardware realization.

As an instructional tool this simulator has proven very useful. It allows students to prepare architecture designs at a block level without specifying the timing details of a microcoded machine. It also provides a solid experience in dealing with the issues of parallelism and speed versus hardware quantity and cost which are at the heart of many architectural decisions.

The use of simulators for instruction in microprogramming is common practice. Robert Rosin used microcode simulators running on the CDC6600 in his classes at SUNY Buffalo before moving to specialized machines such as the QM1 [9]. Mathur [8] has implemented a simulator, MICROSIM, which simulates the structure of the Interdata 70, a microprogrammable architecture. MICROSIM is used for teaching microprogramming and emulation. Simulators of this type have a relatively fixed hardware configuration but do allow for experimentation with the emulation of different target machine instruction sets. Multiple levels of microcode can provide more flexibility, but do not allow for realistic hardware design experimentation.

At an even more abstract level, most architecture references use an instruction-set processor language such as ISPS [1] to describe instruction set meanings. Interpreters are available for these languages. The advantage of such systems is that they allow the description of an instruction set which is more or less implementation independent—the first step in the design process.

Efforts have also been made to develop structured higher-level languages which produce efficient microcode [5]. They have been only moderately successful. The structure of the language used here is closer to that of an assembler. Assemblers for microcoded machines are not easy to use because each instruction has the capability of specifying many independent and concurrent operations. Still, they are frequently used because of their power, and are easy to construct in FORTH [2].

As a hardware development tool the Wright State simulator is particularly useful for testing basic hardware organization concepts and making instruction set choices. The ability to perform test runs using the microcode not only supports the correctness of the microcode but also verifies the basic correctness of the hardware organization chosen, thus eliminating many of the costly errors associated with prototype hardware implementations. In addition, certain microcode sequences may be identified as particularly costly, and suitable adjustments can be made to the hardware design and instruction set. Forsley [6], reviewing RISC and Forth machines, suggests that simulation of new machines is a reasonable approach to obtaining performance expectations. Applying the code generator to tested microcode simplifies the process of generating the binary image of the microcode for a hardware implementation.

The FORTH Microcode Simulator (FMSIM) and FORTH Microcode Generator (FMGEN) are implemented in Laboratory Microsystems Z-80 FORTH V3.0, running under CP/M [4]. FMSIM occupies some 55 screens and FMGEN occupies another 31. When loaded, FMSIM requires 10K of dictionary space and FMGEN requires 5K. Most words used are FORTH 83 Standard. The FORTH environment has proven to be very appropriate for the development and use of these tools. Developed by students on small microprocessors, the simulator and code generator form a compact and simple-to-use package. Simulations can be performed in a convenient, interactive fashion for relatively little cost in terms of computing capacity. The use of FORTH also makes these tools easily modifiable and extensible [3]. The use of special, executable FORTH mnemonic microcode, rather than the generation and interpretation of binary microcode, makes the simulator more compact, efficient, and easy to use [11].

The following sections describe the simulator and code generator in more detail. Use of these tools is also described. A complete example is included in the Appendix.

2. *Microcode Simulator - Overview*

The first step in designing a microcoded system using FMSIM is to prepare a block diagram similar to the partial diagram shown in Figure 1(a). The user must then translate this diagram into a hardware description for FMSIM by first declaring each hardware device used in the diagram. FMSIM provides FORTH defining words for declaring the following types of devices: `BUS`, `REGISTER`, `POINTER`, `DRIVER`, `ALU`, `MEMORY`, `PROM`, `ADDER`, `INCREMENTER`, `MUX`, and `COMPARATOR`. A FORTH programmer can easily add extra device types if required.

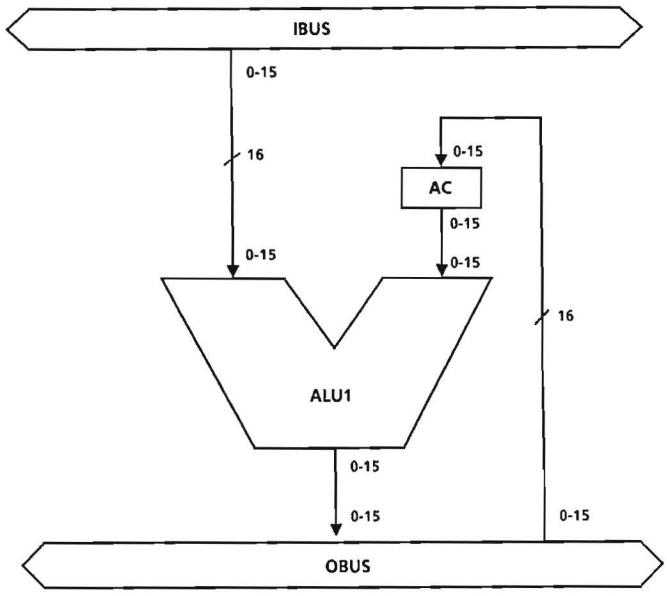

Figure 1. (a) Simple block diagram of an ALU and accumulator.

```
!! OBUS          << 0 :: 15 >>  <-- ALU1   << 0 :: 15 >>
!! ALU1 PORTA    << 0 :: 15 >>  <-- IBUS   << 0 :: 15 >>
!! ALU1 PORTB    << 0 :: 15 >>  <-- AC     << 0 :: 15 >>
!! AC            << 0 :: 15 >>  <-- OBUS   << 0 :: 15 >>
```

Figure 1. (b) Connection statements for the diagram of 1(a).

The FMSIM declaration of each device includes an instance name, the device type, and any modifiers needed to specialize the operation of the device. For example:

```
3.STATE REGISTER AC
```

defines a register with name `AC` and tri-state outputs. For each device declarations FSIM creates a FORTH definition containing a special data structure which represents the device.

After all hardware elements have been declared, the user then specifies the primary data path connections among them. For example, the statement:

```
!! AC << 0 :: 11 >> <-- OBUS << 0 :: 11 >>
```

specifies that inputs `0` to `11` of the accumulator `AC` are connected to the bus `OBUS` in corresponding order. Connection statements modify the previously defined device data structures, filling in information that is used to update the states of these devices when the simulation is running. Figure 1(b) illustrates the connection statements for the diagram in Figure 1(a). Note that connections are only made for data paths. All control is implicit in the device declarations and in the microcode written.

At this point the user has described the hardware organization of some desired architectural implementation. The next step is to describe the microcode instructions that will control the sequencer, the ALU, and all other components of the system. In the case where this system interprets another higher level instruction set, this microcode, together with the defined hardware, provides the instruction set processor (ISP) for the target machine [1]. The user is allowed to define names for certain bits, allowing convenient mnemonic specification. All addresses in the code are specified by `LABEL`s, which are also words which must be defined. The user then writes microcode using

these names, the names of the hardware devices previously defined, and a set of microinstuction operations. Two operations are the most common: **ENB** (enable) and **LD** (load). **ENB** enables the output of tri-state devices, and **LD** is used to signal the gating of information into a device. **ALUs**, of course, have a more extensive set of operations which sometimes require modifiers. The general format for microinstruction components is:

```
<operation> <device_name> <modifiers>.
```

For example, the micro-operations:

```
ENB MA
SEL ALU1 F=B
LD RAM
```

specify the following actions: enable the memory address register, select the "pass B" function at **ALU1**, and load **RAM** with the data on its inputs.

Microinstructions are composed of sets of these micro-operations (describing the data transfers occurring in one cycle of execution) terminated by sequence control operations which tell the microsequencer which microinstruction to execute next. For example, **CONT** specifies the end of one microinstruction and passes control to the following one. **JMP FETCH** ends one micro-instruction and passes control to the microinstruction with label **FETCH**. For example, the following two microinstructions:

```
[ DCA ] ENB MA SEL ALU1 F=B ENB ALU1 LD RAM   CONT
        SEL ALU1 F=Ø  ENB ALU1  LD  AC   JMP FETCH
```

perform the "deposit and clear accumulator" assembly instruction for the implementation of the PDP-8 described at the end of this paper. **[DCA]** is a label for the first instruction.

One feature of this simulator is that the microsequencing component of a simulation is implicit in the simulator. In other words, the user does not specify sequencing hardware. Operations such as **CONT**, **JMP**, and **JSR** are provided. Further, the system provides the user with a special sequence control operation, **JVCTR**, which is used in decoding the machine instructions of the simulated machine. Usually the user includes a mapping **PROM** as part of the hardware design. This takes the instruction opcode as an address and converts it into an entry point into the microcode. The decode procedure (which the user must include) sends this micro-address to the sequencer via a predefined simulator device called the **MPC**. The **JVCTR** operation causes a branch to the address at the **MPC**, executing the instruction.

After constructing microcode for the hardware implementation defined, the user loads the simulator screens into the dictionary. Then the screens containing the hardware and microcode descriptions are loaded. As these screens load, data structures are created for the hardware devices and connections. The microcode description compiles into one large, executable FORTH word. To execute the simulation, the user types **RUN <label>**, where **<label>** is the desired entry point. As each microcode statement executes the states of the data stuctures change (with the simulated data transfers). The simulation will continue to execute freely until the end of the definition is reached or until an error condition is encountered.

Currently, the only statistic kept is the total number of microinstructions executed. Addition of other system statistics as desired is a simple matter. The simulator provides a number of control words which allow the user to start the simulation at various points, set breakpoints, single-step (trace) the execution, and examine and modify the contents of various devices in the simulation. This can be done both interactively and by the creation of a simulation control program.

The next sections describe in greater detail the structure and operation of the simulator program. A complete specification of the simulation language is also provided. The reader may wish to consult the example given in the Appendix at the end of this paper for a better understanding of what the user must provide for the simulation.

3. FMSIM Structure and Operation

The simulator can be divided into four functional units. The first unit contains the code which defines system data structures corresponding to the hardware devices declared by the user. The second unit is responsible for parsing connection statements and for generating and filling in the data structures necessary for updating the simulation state as the simulation proceeds. The third unit consists of the definitions which provide the user with the microcode operations which are used to specify the microcode and which, when executed, drive the simulation. Finally, the fourth unit provides the simulation control words which allow the user to interact with the simulation. Each of these units is described in detail below.

3.1 Hardware Definition

Figure 2 shows the types of devices available on the simulator and the device definition syntax. Note that BUSses are treated as hardware devices by the simulator. REGISTERs, which are latched, have one input and one output, and may be either 2-state or 3-state devices. DRIVERs are always 3-state, and are treated much like registers, with the exception of having no memory. A POINTER is a special register intended to be used for addressing stack memories. It includes increment and decrement operations that are used for PUSH and POP instructions. An INCREMENTER is a non-storage device which simply transfers the incremented input to the output. A MUX is a multiplexer, which is a non-storage device with any number of sets of inputs. Three types of two-input devices are provided. ADDERs perform the expected addition operation. ALUs have several different functions which may be selected; these are discussed later. COMPARATORs test their inputs for equality and produce a one bit output which can be used to control conditional branching in the microprogram.

Two kinds of memory devices are provided. MEMs are equivalent to RAM, and have data inputs, data outputs, and address (MAR) inputs. PROMs are similar, with the obvious exception of having no data inputs. Both of these types come in any length and in three widths: 8-bit, 16-bit, and 32-bit.

All non-memory devices are 32 bits wide. Not all of these bits must be used in connection statements or the simulation if a narrower width is sufficient. REGISTERs, POINTERs, ALUs, and MUXs may be either 2-state or 3-state; all other devices are 3-state by default. As illustrated in Figure 3, each device desired by the user is defined using one of the device defining words provided by FMSIM. Each of these words creates a definition with the name specified by the user, and generates a data structure suited to each type of device. Figure 4 shows the general form of these data structures.

The first two fields are the same for all devices. The data field occupies the first two words; it holds the current state of the device, and is also used as a work area. The next word is the type field, which contains a constant indicating the type of the device.

For most devices the next field contains a pointer to a linked list of output connection nodes. Each of these nodes describes connection information for each bus that is connected to the outputs of the device. The following fields are pointers to linked lists of input connection nodes, one list for each input port on the device. (For example, REGISTERs have only one IN_LIST pointer; a four-input MUX will have four.) The remaining fields contain other device-specific information, such as the number of inputs on a multiplexer or the storage area for memory devices. Bus data structures have no input or output lists, but they do have a link so that the system may keep a master bus list.

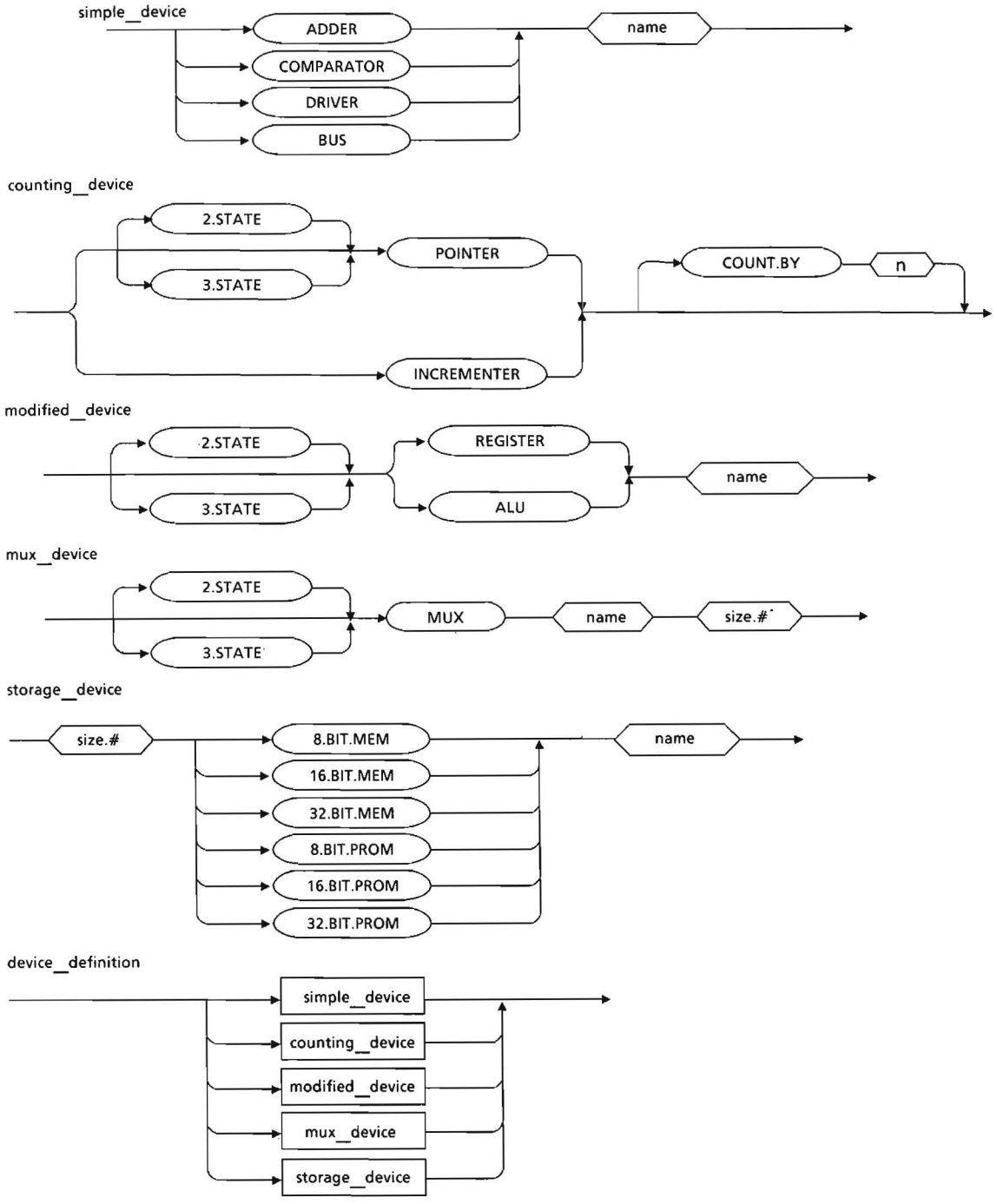

Figure 2. FMSIM devices and device definition syntax.

```
                BUS             IBUS
                BUS             OBUS
3.STATE         ALU             ALU1
2.STATE         REGISTER        AC
```

Figure 3. Device definitions for the diagram of Figure 1(a).

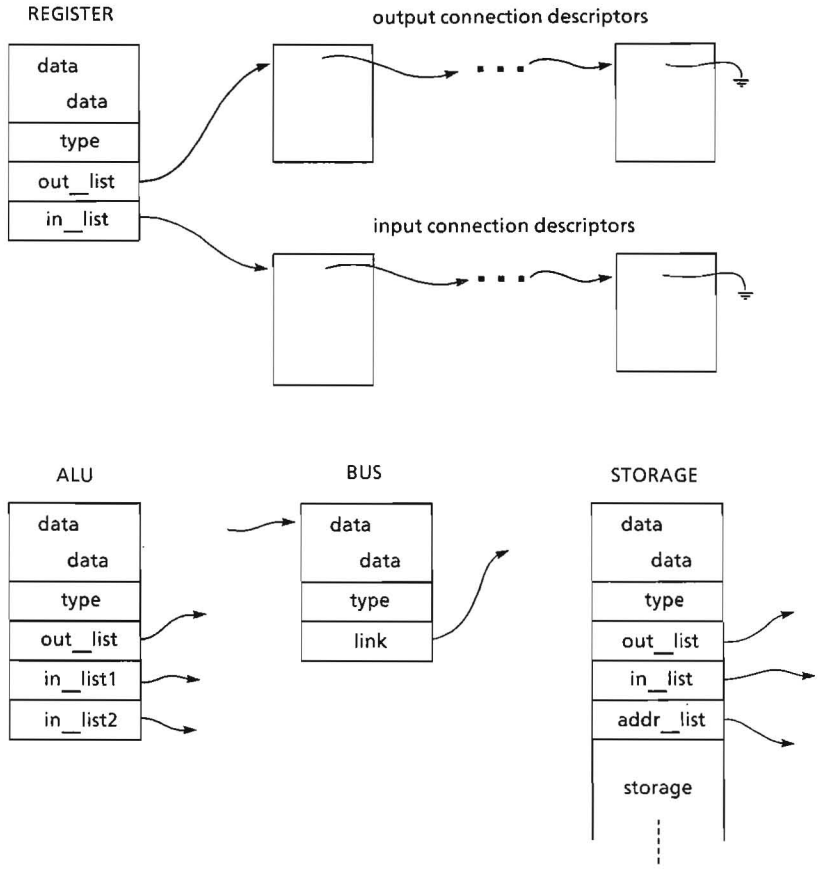

Figure 4. FMSIM device data structures.

3.2 Connection Descriptions

A large portion of FMSIM is devoted to parsing the connection descriptions and generating the corresponding connection node-lists described above. A fairly flexible connection syntax is allowed; this is described by the syntax diagram in Figure 5. (Refer to Figure 1(b) for examples of connection statements.) The general format is as follows: the inputs of some device, qualified by a port selector if applicable, are connected to the outputs of the device described on the right side of the `<--` symbol. Each statement connects a contiguous group of bits at a device output to a contiguous group of bits at a device input. The field lists shown in the syntax diagram describe these bit fields. The input and output bit fields must be the same size, but they do not have to be in the same positions. It may take several statements to describe the wiring at a single port, but it is possible to effect connections in any configuration (one bit at a time, if necessary). If the bit field list is omitted the statement connects the entire width of the device. FMSIM provides special `VCC` and `GND` device definitions for connecting inputs to logic-1 and logic-0 values.

The first element of a connection statement, the ! !, is a simulator word which performs the parsing and node generation for each statement. First, temporary pointers are created which point to the source and destination device words as well as the node list which is to be updated. The bit field descriptions are also saved in temporary variables. Then a new node is allocated. The structure of this node is shown in Figure 6. The link field is used to form node lists. The device field is a pointer to either (a) the destination device data structure, if this node is on an output list; or (b) the source device data structure, if this node is on an input list. The shift field indicates the number of bits left or right that the data from the source must be shifted to align properly with the destination field. The mask field is used to select the desired bits from the source device. The mask and shift values are calculated from the bit-field description and stored in the new node. Error checking for device and bit-field compatibility is also performed at this time. After the node is filled in, it is linked to either the output list of the source device (if the destination is a bus) or the input list of the destination device.

The final part of the hardware description is the set of **EQUATE** statements. These statements declare the bit names that will be tested for conditional branches in the microprogram. **EQUATE** statements may be viewed as wiring status bits into the microsequencer. As indicated in Figure 7 their syntax is similar to that of connection statements. **EQUATE** parses the statement and creates a definition and data structure which contains a pointer to the source device and a mask which selects the desired bit. **EQUATE** defined words may then be used as the objects of **TST** operations in the user-defined microcode.

At this point the hardware implementation is completely specified. The next step is to describe the microcode part of the implementation of the application or ISP.

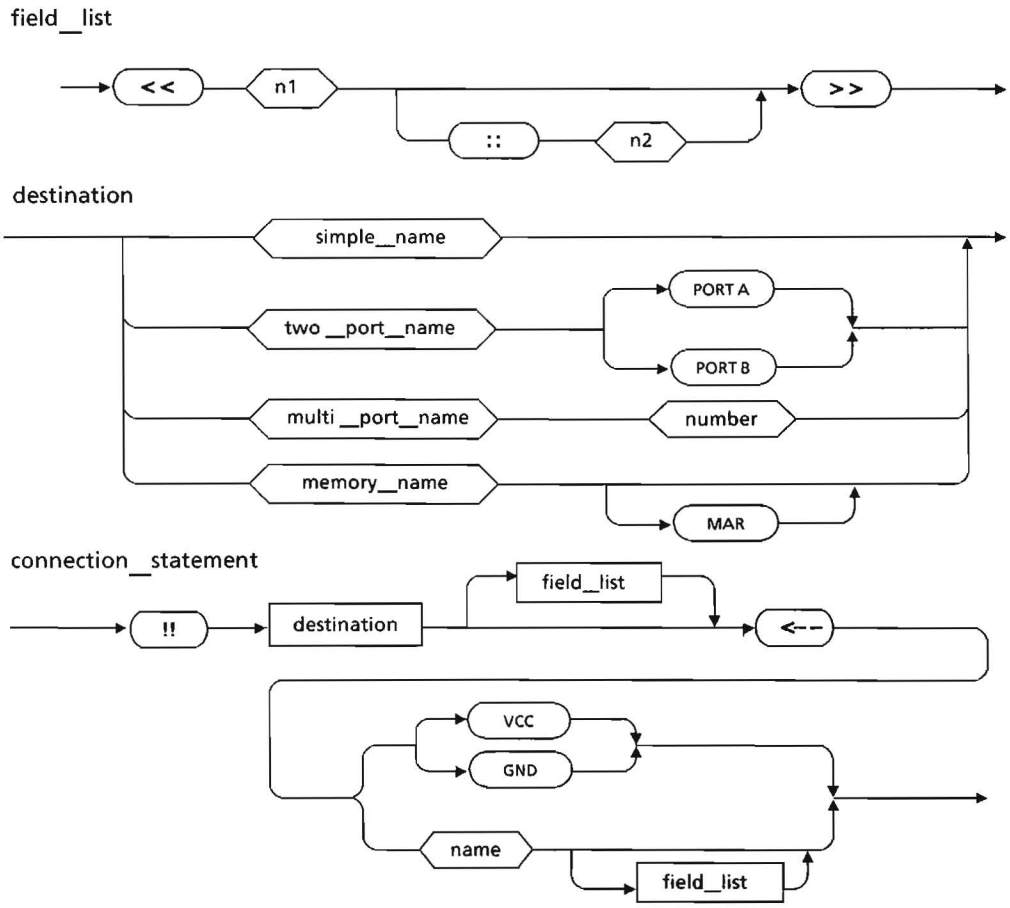

Figure 5. Connection statement syntax.

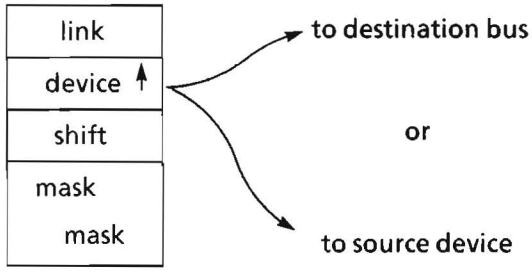

Figure 6. Connection description node structure.

Figure 7. **EQUATE** statement syntax.

3.3 Microcode Operations

In order to define microcode for the simulation the user applies "micro-operations" to the devices and test bits previously defined. These microcode operations are provided by the simulator and are responsible for the actual execution behavior of the simulation. When the microcode definition is compiled into the dictionary, these words execute immediately. Typically, they compile literal pointers to the devices on which they operate, and then compile appropriate run-time actions into the definition. Much of the testing of device types for choosing run-time actions and most of the error checking is performed at compile time, making the simulation more effective and efficient.

As mentioned previously, the user must ensure the proper order of the operations within a simulated microinstruction. FMSIM was developed to simulate microcycles that consist of the following sequence: (a) at the beginning of the cycle selected storage devices are enabled; (b) the enabled data propagates through the system, passing through non-storage devices as required, and establishing new values at the inputs to other storage devices; (c) the new values are loaded into the storage devices, ending the microcycle. Although FMSIM was developed with this sequence in mind, there is nothing in the simulator to enforce this form of microcycle. Micro-operations are executed sequentially, in the order specified. It is thus possible to develop microcode for a more complicated timing and sequencing scheme which will be properly simulated. However, the user must ensure that the micro-operations specified are consistent with the restrictions imposed by the timing scheme chosen.

Micro-operations fall into two categories—those which are concerned with data flow, and those concerned with sequence control. Most operations, such as **LD**, **ENB**, and **SEL**, control the flow of data. The last operation in each microinstruction definition is a sequence control word.

Before examining the data-flow operations it will be useful to describe how FMSIM transfers data between devices during the execution of a simulation. As mentioned previously, a micro-operation compiles two items into the microcode definition. The first is a pointer to a device data structure; the second is the compilation of a run-time procedure. This procedure uses the pointer to

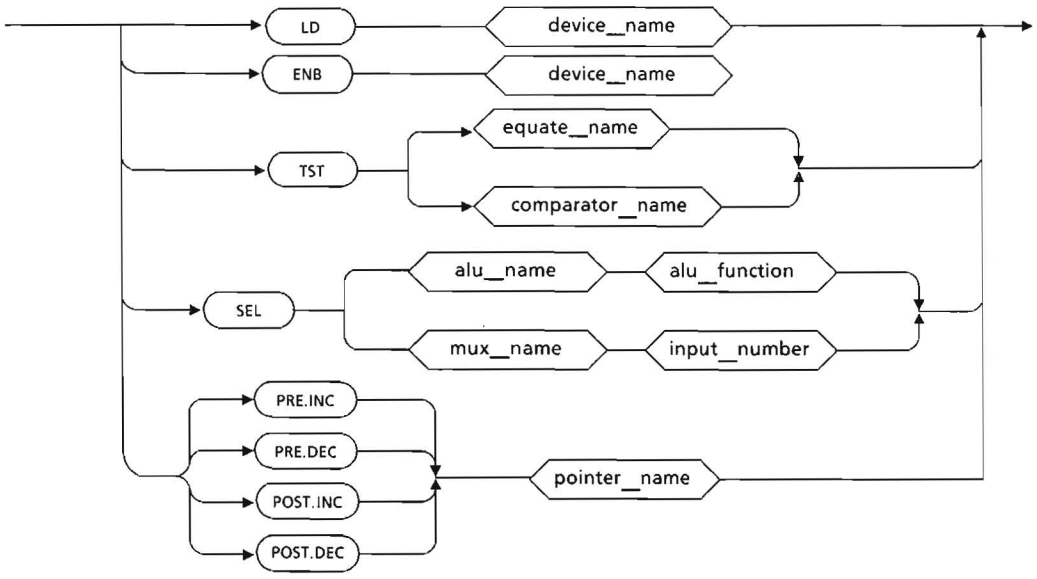

Figure 8. FMSIM data-flow micro-operations and syntax.

obtain from the device data structure a pointer to the appropriate node list. It then processes each node in the list. First, it uses the device pointer to obtain the current value at the source device. Then it uses the mask and shift fields to extract and align the proper group of bits from the source. This is then ORed into the destination value. Processing of output lists is similar.

Figure 8 shows the syntax for the data-flow operations. The **LD** operation processes an input list to load a device with a new value. The **ENB** operation processes one or more input lists to obtain the current value if the device is a non-storage type (e.g. **ADDER**). If the device is a storage type (e.g. **REGISTER**), it obtains the current value from the device data structure, and then distributes this value to the destinations described by the output node lists.

The **SEL** operation may apply to either a **MUX** or an **ALU**. When applied to a **MUX**, **SEL** uses the name and the port number to compile a literal pointer to the proper input list of the named device into the definition. Then **SEL** compiles a load type action into the definition. When applied to an **ALU**, **SEL** compiles a pointer to the **ALU** and the selected run-time function into the definition. These run-time functions process the input node lists of the **ALU** to obtain the current inputs and then perform the desired computation, leaving the results in the device data field. FMSIM provides sixteen **ALU** functions as listed in Figure 9 below.

```
F = 0                    F = A + 1
F = 1                    F = B + 1
F = A                    F = A + B
F = B                    F = A - B
F = -A                   F = B - A
F = -B
                         F = A ^ B (AND)
F = NOT.A                F = A V B (OR)
F = NOT.B                F = A X B (exclusive OR)
```

Figure 9. **ALU** functions.

The TST operation can be applied to EQUATE names or directly to the outputs of comparators. In either case execution sets a condition code variable to either TRUE or FALSE, depending on whether the bit or comparator output is a 1 or 0. This condition code variable is then used by the conditional microsequencing operations described below.

The four operations PRE.INC, PRE.DEC, POST.INC, and POST.DEC apply only to POINTER devices, incrementing or decrementing the current stored value. FMSIM provides these operations as substitutes for up-down counters, simplifying the implementation of such structures as stacks.

Before describing the micro-operations provided to control the microsequencing, it will be useful to describe the way FMSIM handles microcode addresses. Remember that the microcode description compiles into one long FORTH definition. Simulating the sequencing of the microinstructions requires that the interpreter make jumps back and forth to different points within this definition. This is accomplished by using special simulator constructs called labels.

Each label used in the microcode must be defined in advance using the LABEL defining word. The user assigns a predefined label name to a particular microinstruction by prefixing the instruction with the name enclosed in a "left-bracket"–"right-bracket" pair. (See the example in Figure 10.) When the microcode definition is loaded, every label inside the compile state brackets executes immediately, taking the current dictionary address and storing it inside the label body. When a label is encountered as a destination address (outside of brackets) a reference to its dictionary word is compiled into the dictionary. When loading is complete, all necessary dictionary entry points are contained in label definitions. The actual use of these is described below.

FMSIM provides the user with nine different sequence control micro-operations which use these labels to guide the run-time microsequencing of the simulation. Some of them are immediate, as with the data-flow words described above, and some are simply compiled into the definition to execute when the simulation is run. The syntax for these words is specified in Figure 11. The basic mechanism used to control sequencing involves manipulation of the return stack. When execution must be transferred to some other part of the definition, the pointer to the next word in the definition is dropped from the return stack and replaced by the dictionary address extracted from the appropriate label definition. When the microsequencing FORTH word terminates, control is automatically transferred to the desired destination.

Four jump operations are provided. JMP <label> transfers control directly to the labeled microinstruction. JTR <label> and JFS <label> check the condition code modified by TST before performing a jump. JSR <label> saves the current return pointer on a special simulator stack before performing a jump to the label.

Three return operations are provided. All of these work by replacing the current return pointer with the pointer left on the top of the simulator stack by the most recent JSR. RET is an unconditional return from a microcode subroutine. RET.TR and RET.FS check the simulator condition-code flag before executing the return.

```
LABEL        CMLINK
LABEL        TST.RAR
                .
                .
                .
[ CMLINK ]  SEL ALU1  F=NOT.B  ENB ALU1  LD L
            TST IAC   JFS TST.RAR
```

(label marking μ-code entry point)

(label as destination of a -sequencing operation)

Figure 10. Label definition and use.

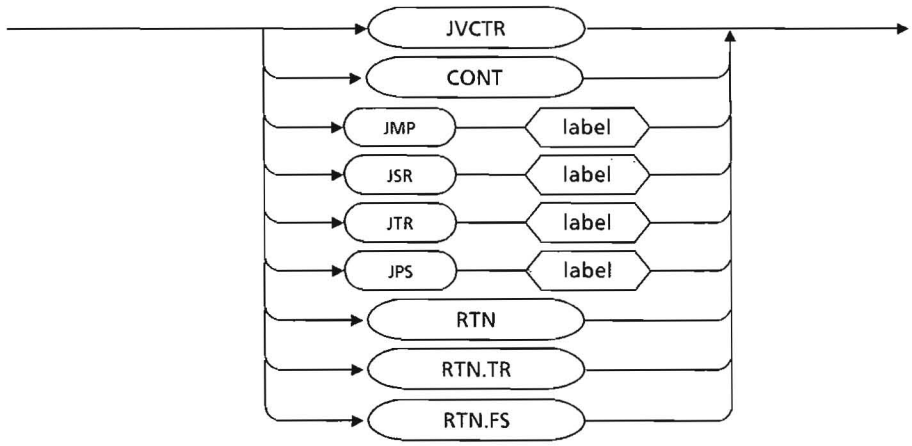

Figure 11. Microsequencing operations and their syntax.

One special microsequencing operation is provided especially for the instruction decoding and microcode entry sequence. `JVCTR` allows the user to jump to a microcode entry point that was specified in a special instruction mapping `PROM`. This `PROM` is programmed by the user using the `LOAD.VCTR <label>` operation, which loads the `PROM` with the address corresponding to the named label. If this vector is enabled onto the special `MPC` bus (already defined by the simulator) when the `JVCTR` is executed, control will pass to the desired microcode entry point.

3.4 User Interface

After completing and loading the description of the hardware and microcode being simulated, the user must specify how the simulation is to be driven. In the typical case of the interpretation of some `ISP`, this usually means loading machine language opcodes into the simulated program memory and initializing the simulated system (the program counter, for example). Because the simulator is set in the FORTH environment, simulations are interactively accessible to the user. However, to allow clean and convenient interaction with the simulation, FMSIM provides a few additional facilities.

`RD <name>` displays the contents of the named device or bus. `<value> WT <name>` places the double word on the stack into the data field of the named device. Memory devices may be loaded or read by the words

```
<data> <address> <name> LOAD.MEM.n
```

and

```
<address> <name> READ.MEM.n
```

respectively, where n specifies the bit width of the memory (8, 16, or 32). `RUN <label>` starts execution of the simulated microcode at the specified label. All of these words may be used both interpretively and as part of a colon definition for simulator control or interactive debugging.

Address vectors for decoding machine instructions or generating microcode addresses can be loaded into storage devices using `LOAD.VCTR <address> <name> <label>`. `LOAD.VCTR` takes the address contained in the label definition and places it in the named storage device at the specified address.

Two counting breakpoints are provided for debugging purposes. `<n> LD.BK1 <label>` and `<n> LD.BK2 <label>` specify the label at which execution should stop and the number of times (n) that the instruction should be executed before halting. The sequencing micro-operation at the end of each microinstruction checks to see if break points are enabled and if the next micro-address is the one specified in the breakpoint. If it is, and if the instruction count matches, then the destination address is left on the simulator stack, and execution returns to the interpreter. At this point the user may examine the simulation statistics, and examine or modify the contents of any simulator device. Execution may be resumed by the command `GO`, which expects the starting address to be on the simulator stack. The breakpoint count is not reinitialized—all subsequent breaks will occur each time the label is encountered unless a new breakpoint specification is entered.

A microtrace debugger facility is also provided. The user specifies the starting location and initializes the microtrace routine by executing `START.AT <label>`. After performing this operation the user can then single-step through the individual operations inside a microinstruction by repeatedly executing the word `MT`. The state of the simulation may be examined between executions—control is returned to the interpreter between micro-operations.

4. FMGEN Structure and Operation

One of the primary differences between FMSIM and many other machine simulators is that microcode written for it executes directly, rather than generating microcode bit patterns which are then interpreted to drive the simulation. This is possible because of the power and flexibility that FORTH provides for defining the simulation environment. While this feature is advantageous for the actual designing and running of simulations, it is at some point desirable to be able to generate horizontal microcode bit patterns from microcode that has already been tested on the simulator. The ability to use the same input description for both simulation and microcode generation reduces errors and labor. FMGEN, the complementary microcode generator, provides this capability.

In FMGEN the device-defining words, microcode operations, and simulation control words are redefined. Instead of constructing an executing simulation definition out of the user's microcode, a definition is constructed, using user-supplied mapping information, which when executed generates the corresponding microcode bit patterns. This microcode is written as ASCII data to specified screens in the disk file, which can then be used as input to tailored PROM-programming systems. FMGEN consists of three functional units: the first unit constructs code-generation data structures from the simulation hardware description; the second unit creates mapping-description data structures corresponding to the mapping information provided by the user; the third unit compiles the microcode description, creating a FORTH definition that will generate the binary microcode. These units are described below.

4.1 Hardware Definition

Each of the device-defining words (`REGISTER`, `ALU`, etc.) used to specify the hardware devices present in the simulation is responsible for the creation of device mapping definitions. These definitions are data structures which contain pointers to lists, each of which describes the desired bit-mapping which corresponds to one of the possible operations on that device. For example, `REGISTER <name>` defines a word with the structure shown in Figure 12(a). Note that there are only two allowed register operations—`LD` and `ENB`. Each of the lists is made up of mapping descriptor nodes which specify which bits or bit fields are to be set when that operation (`LD R1` for example) occurs. The next section describes the structure of these mapping descriptor-nodes.

Figure 12(b) describes the allowed operations and the corresponding data structures for each of the other device types. Note that `BUS`ses do not have any operators, and thus are ignored (null definitions). `EQUATE` names are treated as objects of the `TST` operation, and are thus given definitions similar to `COMPARATOR`s. `ALU`s are currently implemented with sixteen possible functions (and `ENB`), requiring a corresponding number of mapping lists.

In addition to creating data structures for the named devices, these defining words also specify the runtime action which occurs when the microcode definition executes. In general terms, each of these words processes the appropriate mapping list, inserting bit patterns into the binary microword currently being assembled.

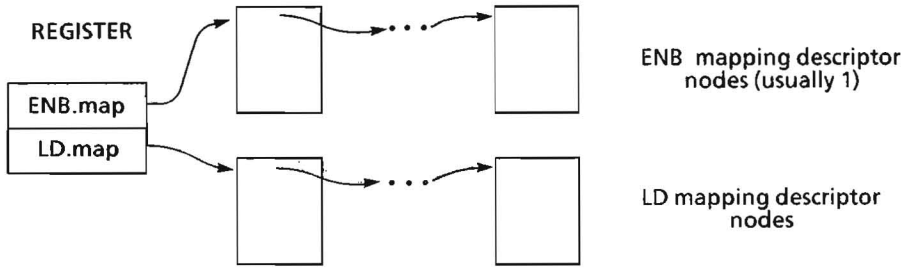

Figure 12. (a) Device mapping data structure for **REGISTER** (general).

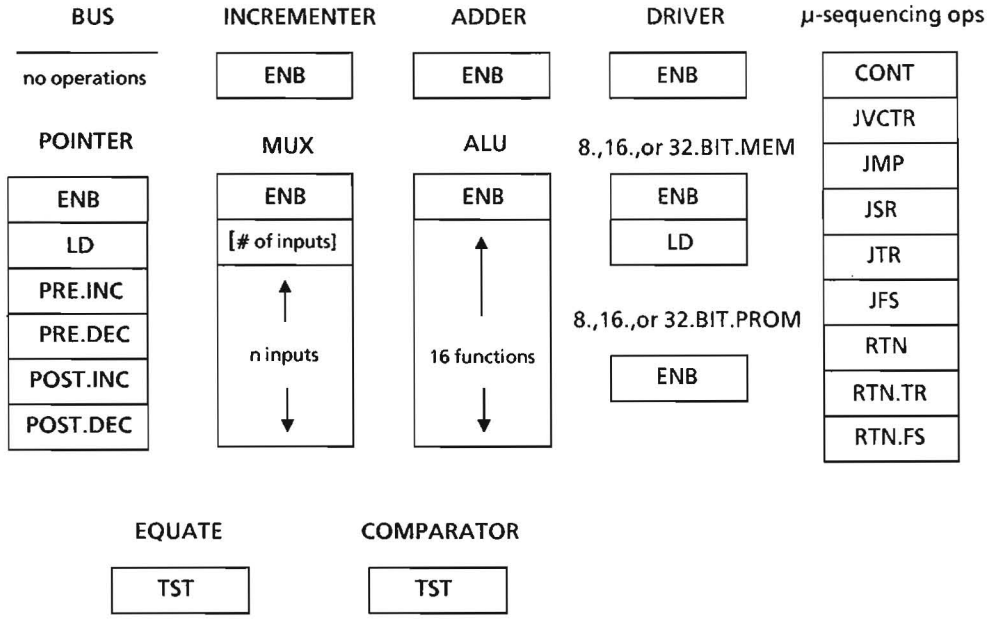

Figure 12. (b) Operation mapping lists for other devices.

4.2 Mapping Instructions

In order to generate bit patterns in the binary microword from the device descriptions defined above and the generation instructions in the simulation microcode, some form of mapping information is necessary. This mapping information indicates which bits and/or fields in the microword must be set to signify the desired operation. Once this information is provided it can be placed in the mapping descriptor nodes on the appropriate operation list for the device, as described above.

After defining the simulated hardware and before defining the microcode, the user must specify the bit mapping for each possible combination of device and operation present in the simulation. This is accomplished using the mapping definition words **FIELD**, **AT**, and **IGNORE** provided by FMGEN. The complete syntax description for these words is given in Figure 13.

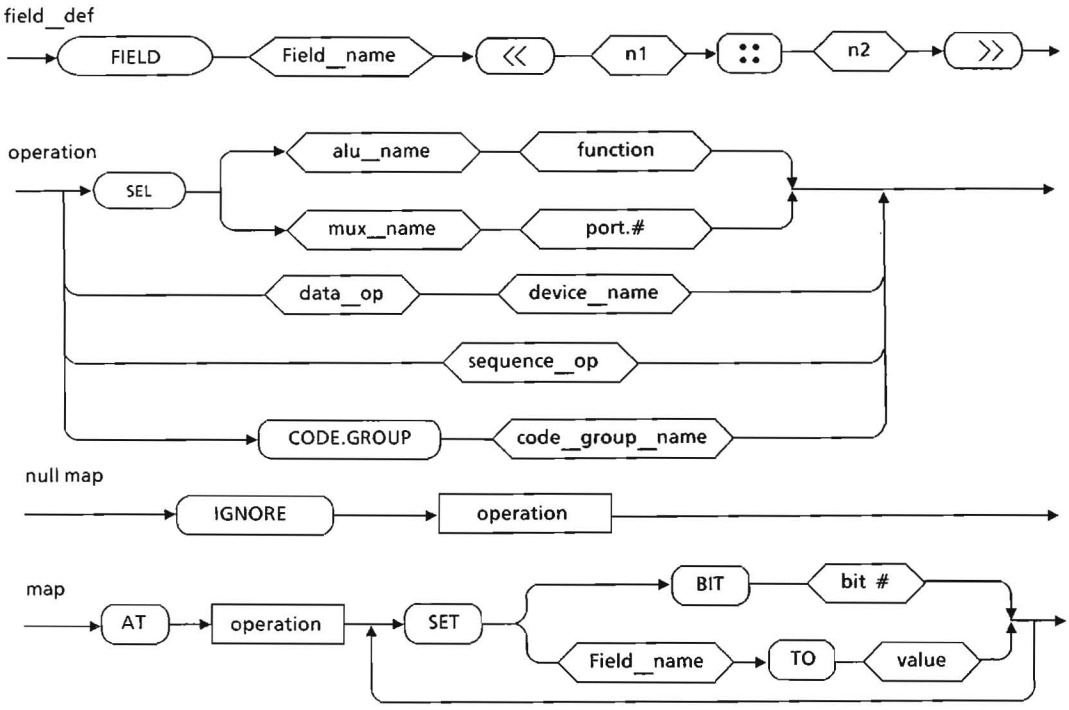

Figure 13. Mapping statement syntax.

The first thing that must be specified is the size (in bits) of the microcode word being generated. A maximum size of 232 bits is allowed. **<n> MWORD.SIZE** sets the microword width.

Next, for notational convenience, the user specifies contiguous groups of bits within the microword and assigns names to them. This is done using the **FIELD** statement shown in the diagram. Note that the bit field description is the same type used by connection statements. The word **FIELD** is a defining word which parses the statement and creates a named definition containing the bit numbers. This definition has a run-time action which is executed when the field name is used in a mapping statement. This is described below.

The **AT** statement is the primary mapping description statement. In the example below,

```
AT  LD AC  SET BIT 33
```

AT specifies that the LD AC operation is signified by setting bit 33 of the binary microword. As can be seen from the diagram in Figure 13, AT statements can describe mapping for data flow micro-operations, sequence control micro-operations, and CODE.GROUP specifications. The CODE.GROUP specification allows the user to map sets of recurring operations as single items. Code groups for which no mapping specification is given are mapped according to their individual components.

AT uses the specified operation to generate a pointer to the corresponding node list in the device data structure. SET then allocates a new mapping node and attaches it to the end of the list, returning a pointer to it. The structure of this node is shown in Figure 14. Mapping description nodes come in two types. Numeric nodes contain a link for list maintenance, a byte offset into the microinstruction assembly area, and a 16-bit mask for setting the desired bits. Label nodes are used for mapping micro-addresses. They also contain a link field. However, instead of containing the actual bits to be mapped (which are unknown at mapping time), they contain an indicator as to whether the label address is to be used as an absolute or a relative address. The third word contains a pointer to the field descriptor of the address field.

When a BIT is being set, BIT consumes the following bit number and uses it to generate an offset and mask in the mapping descriptor node. When a FIELD is being set to some value, FIELD executes and consumes the rest of the statement. If the word after TO is a number, then another numeric node is generated with offset and mask. If the word is an addressing type descriptor, then a label node is constructed with the type specified and a pointer to the address field.

LABELs are also defined differently from the simulator. LABEL is a defining word which creates a place to store the line count which is current when that label executes. The contents of labels are used when the label mapping nodes are executed.

Connection statements play no role in the generation of the microcode, and are skipped over by FMGEN (null definitions). It should be mentioned here that the mapping information is treated in a similar manner by the simulator, allowing the exact same screens to be used as input to both FMSIM and FMGEN.

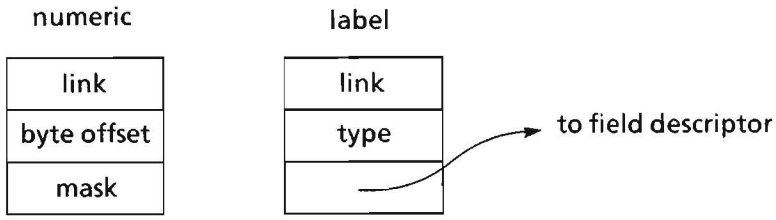

Figure 14. Mapping description node structure.

4.3 Micro-Operations and the Microcode Definition

As mentioned previously, the hardware and microcode descriptions compile into a FORTH definition which is then executed to generate the binary microcode. Data flow operators such as LD and ENB compile as constant offsets which select the proper mapping list in each device data structure. The device names are compiled as normal definitions with the run-time action of processing one of the node lists in the device body. Micro-sequencing operations such as CONT and JMP are immediate words which perform two actions at compile time. First, they increment the microcode line counter. This allows labels (which are also executed during compilation) to be loaded

with the proper micro-store addresses before execution, eliminating the problem of forward references. Then they compile the appropriate run-time procedure into the definition.

In order to run the generator definition, the user must specify the starting screen where the ASCII microcode is to be stored using the statement `<n> START.SCREEN`. The statement `GEN.CODE <name>` then initializes the line counter and the pointers used to output the data, and executes `<name>`. As each micro-operation is executed, its corresponding mapping node-list is processed. This involves moving through the list and masking bits into the microinstruction at the proper offset. At the end of each microinstruction the node-list corresponding to the sequencing operation is processed. If the operation is a jump to some label, this involves processing a label node. If absolute addressing is indicated the contents of the destination label are simply masked into the specified address field. Relative addressing requires the calculation of an offset from the current line counter.

After the entire microinstruction has been processed, the line counter and assembled microword are sent to the output screen, the pointers and line counter are updated, and execution proceeds to the next microinstruction. This sequence continues until the end of the microcode definition.

5. *An Example*

In order to give the reader a better feel for what is involved in using the simulator, and what kinds of simulations are possible, the complete input listing for a simulation of the DEC PDP-8 architecture is presented in the appendix. This simulation is the implementation in microcode of the ISP-level description of the PDP-8 presented in Siewiorek, Bell, and Newell [10]. The reader is encouraged to review this description.

The first step taken by the designer of the simulation is to lay out a block-level description of the hardware implementation of the architecture. Note that in the diagram the only things specified are devices and primary data paths. Control is implicit in the simulation, although the authors suggest that the designer have explicit timing diagrams in mind before constructing microcode definitions.

Screens 21 and 22 contain the hardware definitions for the simulation. Screens 23 through 26 list the connections. Screens 27 through 29 contain `EQUATE`-defined bit names for use in the `TST` operations.

Screens 30 through 35 contain the mapping information necessary to generate microcode for this design. Note that the microword width is 44 bits. Also note the `FIELD` definitions. The `ENCODE.ALU.SEL` statement is a shorthand for encoding the `ALU` function selects as sequential values into the `ALUF` field.

Screen 36 contains the `LABEL` definitions for the microcode which follows. `<n> LABELS` simply applies `LABEL` to the `<n>` label names following.

Line 10 of screen 37 is the start of the microcode definition. This definition continues through screen 44. For readability, the microcode has been organized so that, where possible, there is one microinstruction per line.

Screen 45 contains the operations necessary to "program" the instruction decoding `PROM` named `MAP`. The contents of each of the labels are deposited in `MAP` starting at address `0`, in direct correspondence with the machine opcodes which are used to address `MAP`.

For convenience in testing, a small PDP-8 assembler was written and is contained in screens 46 through 50. This assembler is used to assemble and load the small test program (screens 51 through 53) into the simulated RAM. Note that line 13 of screen 53 loads the beginning address of the program into the PC. When these screens are loaded, the simulation is ready to run, and a `RUN FETCH` command will cause the program to execute.

The test program is a short I/O program which reads a character from the keyboard and prints it ten times. Entering `Q` stops the execution. Although execution time per micro-instruction is not a particularly useful statistic (since it varies with the complexity of the simulation and the host machine used), it does give a general idea of the capability of the simulator. Running the above simulation of the PDP-8 and the test program described on a 4MHz Z-80, the simulation executed

some 4023 microinstructions in 59 seconds, yielding an execution rate of a little over 68 microinstructions per second. While not exactly blinding speed, this is fast enough for useful testing.

Screen 54 contains the commands necessary for generating horizontal microcode for this simulation, assuming the code generator and description file have been loaded. The actual microcode generated is contained in screens 55 through 58.

6. *Conclusion*

FMSIM has been used by some thirty students over the past year in a two-course, graduate-level sequence in computer architecture. These students implement some relatively simple ISP, such as the PDP-8, as a first project and an introduction to simulation with FMSIM. They then work on individual projects of much greater complexity, typically involving the design and simulation of a machine to implement the ISP for a custom instruction set, constrained by some basic implementational approach, such as the use of a stack machine. A previous background in FORTH is not necessary; the minimal FORTH knowledge required to access and run the simulation is easily picked up by the students. However, a knowledge of FORTH is a definite advantage in using the system, particularly for those students who wish to make custom modifications to the simulator or who wish to perform more intricate simulations.

FMSIM is also being used in a continuing research effort aimed at designing and implementing a RISC-type machine to implement a generalized FORTH environment, and is the focus of one related thesis. This project and the preliminary simulation results will be described in another paper. At this time FMSIM has been very successful in helping to verify several design concepts.

Both FMSIM and FMGEN are prototype systems, under constant revision and enhancement, and are by no means production software. No real attempts have been made to optimize the FORTH code, either to minimize memory usage or execution time, or to provide a clean, maintainable structure. A year of use has brought out many shortcomings of the system. In particular, the inability to use data directly as control has proven to be a major limitation on the realism of the simulations. New versions of FMSIM and FMGEN, with various corrections and enhancements, are currently the topic of a thesis which should be completed by March, 1986. Parties interested in obtaining these revised packages should contact the authors.

References

[1] Barbacci, "An Introduction to ISPS", *Computer Structures: Principles and Examples*, Siewiorek, Bell, and Newell, McGraw-Hill, 1982, Chap. 4, pp. 23-32.

[2] Cholmondeley, Gregory, "A FORTH Based Micro-Sized Micro Assembler", *Forth Dimensions*, Vol. III, No. 4, pp. 126-127.

[3] DeWitt, David J., "Extensibility - a New Approach for Designing Machine Independent Microprogramming Languages", *MICRO9 PROCEEDINGS*, Sept. 1976, pp. 33-41.

[4] Duncan, Ray, Z-80 FORTH Version 3.0, Laboratory Microsystems, Los Angeles, California.

[5] Eckhouse, Richard, "A High Level Microprogramming Language (MPL)", *AFIPS*, Vol. 38, 1971, pp. 169-177.

[6] Forsley, Lawrence, "A Review of RISC and Forth Machine Literature", *The Journal of Forth Application and Research*, Vol. 2, No. 4, 1984, pp. 85-86.

[7] Hopkins, William C. and Gary Davidian, "A Micro-programmed Implementation of an Architecture Simulation Language", *SIGMICRO NEWSLETTER*, Vol. 8, No. 3, Sept. 1977, pp. 37-46.

[8] Mathur, Francis P., "MICROSIM: A Microinstruction Simulator for Teaching Micro-programming and Emulation", *SIGMICRO NEWSLETTER*, Vol. 8, No. 3, Sept. 1977, pp. 109-118.

[9] Rosin, Robert, "An Environment for Research in Micro-programming and Emulation", *CACM*, Vol. 15, No. 8, Aug. 1978, pp. 748-760.

[10] Siewiorek, Daniel P., C. Gordon Bell, and Allen Newell, *Computer Structures: Principles and Examples*, 1982, pp. 125-128.

[11] Wyckoff, Richard, "A FORTH Simulator for the TMS 320 IC", *Rochester Forth Applications Conference*, 1983, pp. 141-150.

Manuscript received June 1985.

Alan J. Cotterman received his B.S. in Computer Engineering from Wright State University in 1984, and is currently pursuing an M.S. in Computer Engineering. His interests include computer architectures, robotics, software engineering, and computer vision and image analysis.

Robert D. Dixon received his doctorate from Ohio State University in 1962. He is currently a professor of Computer Science at Wright State University. His interests include real-time systems, and hardware and software support systems for natural language interfaces.

Russ C. Grewe received his B.S. in Computer Engineering from Wright State University in 1985. He currently works for Wright State University under contract with the United States Air Force. His interests include computer architectures, expert systems, network communications, and robotics.

Gerald Simpson received a B.S. in Civil Engineering from the University of Dayton in 1968 and an M.S. from the University of Illinois in 1970. He currently works for the United States Air Force, and is pursuing an M.S. in Computer Engineering at Wright State University. His interests include computer architectures and functional languages.

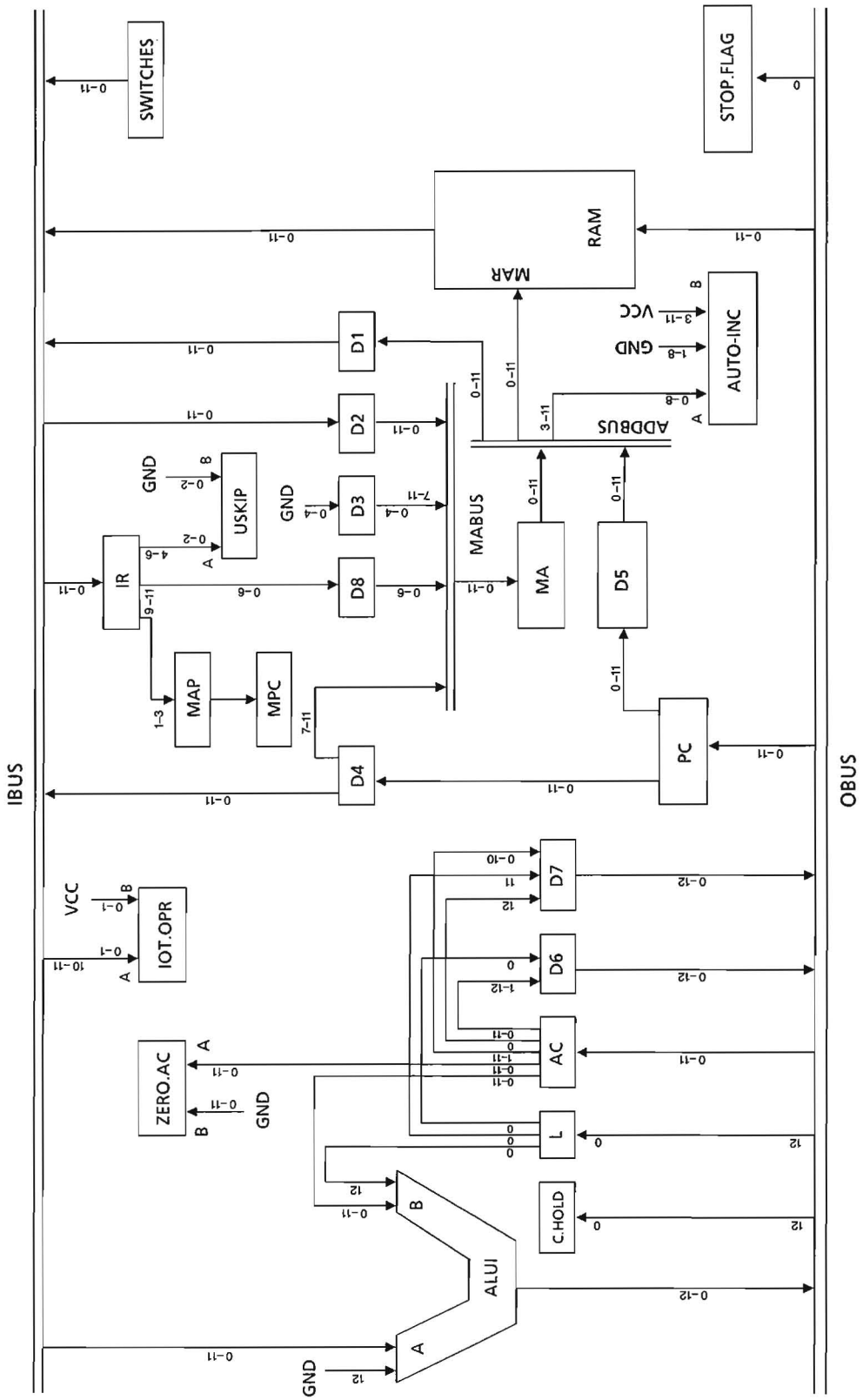

Figure 15. Hardware Diagram
Simulation of PDP-8

Appendix: PDP-8 Simulation

```
Screen # 21
  0 ( PDP-8 ISP SIMULATION )
  1
  2 ( device declarations )
  3              BUS          IBUS
  4              BUS          OBUS
  5              BUS          ADDBUS
  6              BUS          MABUS
  7    2.STATE   REGISTER     AC
  8    2.STATE   REGISTER     L
  9    2.STATE   REGISTER     C.HOLD
 10    2.STATE   REGISTER     PC
 11    2.STATE   REGISTER     IR
 12    3.STATE   REGISTER     MA
 13    2.STATE   REGISTER     STOP.FLAG
 14    3.STATE   REGISTER     SWITCHES
 15    3.STATE   ALU          ALU1                                    -->

Screen # 22
  0 ( declarations continued )
  1
  2 32          16.BIT.PROM   MAP
  3 4K          16.BIT.MEM    RAM
  4             COMPARATOR    AUTO.INC
  5             COMPARATOR    IOT.OPR
  6             COMPARATOR    ZERO.AC
  7             COMPARATOR    USKIP
  8             DRIVER        D1
  9             DRIVER        D2
 10             DRIVER        D3
 11             DRIVER        D4
 12             DRIVER        D5
 13             DRIVER        D6
 14             DRIVER        D7
 15             DRIVER        D8                                      -->

Screen # 23
  0 ( wiring connections )
  1 !!  RAM             << 0 :: 11 >>  <--  OBUS     << 0 :: 11 >>
  2 !!  RAM MAR         << 1 :: 12 >>  <--  ADDBUS   << 0 :: 11 >>
  3 !!  RAM MAR         <<       0 >>  <--  GND
  4 !!  IBUS            << 0 :: 11 >>  <--  RAM      << 0 :: 11 >>
  5 !!  ADDBUS          << 0 :: 11 >>  <--  D5       << 0 :: 11 >>
  6 !!  ADDBUS          << 0 :: 11 >>  <--  MA       << 0 :: 11 >>
  7 !!  D1              << 0 :: 11 >>  <--  ADDBUS   << 0 :: 11 >>
  8 !!  IBUS            << 0 :: 11 >>  <--  D1       << 0 :: 11 >>
  9 !!  PC              << 0 :: 11 >>  <--  OBUS     << 0 :: 11 >>
 10 !!  D5              << 0 :: 11 >>  <--  PC       << 0 :: 11 >>
 11 !!  D4              << 0 :: 11 >>  <--  PC       << 0 :: 11 >>
 12 !!  MA              << 0 :: 11 >>  <--  MABUS    << 0 :: 11 >>
 13 !!  MABUS           << 7 :: 11 >>  <--  D3       << 0 ::  4 >>
 14 !!  MABUS           << 0 ::  6 >>  <--  D8       << 0 ::  6 >>
 15                                                              -->
```

```
Screen # 24
  0 ( connections continued )
  1
  2 !!  MABUS            << 7 :: 11 >>  <--  D4        << 7 :: 11 >>
  3 !!  D3               << 0 ::  4 >>  <--  GND
  4 !!  MAP MAR          << 1 ::  3 >>  <--  IR        << 9 :: 11 >>
  5 !!  MPC              << 0 :: 15 >>  <--  MAP       << 0 :: 15 >>
  6 !!  ALU1 PORTA       << 0 :: 11 >>  <--  IBUS      << 0 :: 11 >>
  7 !!  ALU1 PORTA       <<      12 >>  <--  GND
  8 !!  ALU1 PORTB       << 0 :: 11 >>  <--  AC        << 0 :: 11 >>
  9 !!  ALU1 PORTB       <<      12 >>  <--  L         <<       0 >>
 10 !!  OBUS             << 0 :: 12 >>  <--  ALU1      << 0 :: 12 >>
 11 !!  AC               << 0 :: 11 >>  <--  OBUS      << 0 :: 11 >>
 12 !!  D2               << 0 :: 11 >>  <--  IBUS      << 0 :: 11 >>
 13 !!  MABUS            << 0 :: 11 >>  <--  D2        << 0 :: 11 >>
 14 !!  IBUS             << 0 :: 11 >>  <--  SWITCHES  << 0 :: 11 >>
 15                                                              -->


Screen # 25
  0 ( connections continued )
  1
  2 !!  IR               << 0 :: 11 >>  <--  IBUS      << 0 :: 11 >>
  3 !!  L                <<       0 >>  <--  OBUS      <<      12 >>
  4 !!  D6               <<       0 >>  <--  L         <<       0 >>
  5 !!  D6               << 1 :: 12 >>  <--  AC        << 0 :: 11 >>
  6 !!  OBUS             << 0 :: 12 >>  <--  D6        << 0 :: 12 >>
  7 !!  D7               << 0 :: 10 >>  <--  AC        << 1 :: 11 >>
  8 !!  D7               <<      12 >>  <--  AC        <<       0 >>
  9 !!  D7               <<      11 >>  <--  L         <<       0 >>
 10 !!  OBUS             << 0 :: 12 >>  <--  D7        << 0 :: 12 >>
 11 !!  D8               << 0 ::  6 >>  <--  IR        << 0 ::  6 >>
 12 !!  IBUS             << 0 :: 11 >>  <--  D4        << 0 :: 11 >>
 13 !!  STOP.FLAG        <<       0 >>  <--  OBUS      <<       0 >>
 14 !!  C.HOLD           <<       0 >>  <--  OBUS      <<      12 >>
 15                                                            -->


Screen # 26
  0 ( connections continued )
  1
  2 !!  AUTO.INC PORTA << 0 ::  8 >>  <--  ADDBUS   <<  3 :: 11 >>
  3 !!  AUTO.INC PORTB <<       0 >>  <--  VCC
  4 !!  AUTO.INC PORTB << 1 ::  8 >>  <--  GND
  5 !!  IOT.OPR  PORTA << 0 ::  1 >>  <--  IBUS      << 10 :: 11 >>
  6 !!  IOT.OPR  PORTB << 0 ::  1 >>  <--  VCC
  7 !!  ZERO.AC  PORTA << 0 :: 11 >>  <--  AC        <<  0 :: 11 >>
  8 !!  ZERO.AC  PORTB << 0 :: 11 >>  <--  GND
  9 !!  USKIP    PORTA << 0 ::  2 >>  <--  IR        <<  4 ::  6 >>
 10 !!  USKIP    PORTB << 0 ::  2 >>  <--  GND
 11
 12
 13
 14
 15                                                            -->
```

```
Screen # 27
  0 ( give names to device bits that will be tested )
  1
  2 EQUATE  IB       <--  IR          << 8 >>
  3 EQUATE  PB       <--  IR          << 7 >>
  4 EQUATE  CLL      <--  IR          << 6 >>
  5 EQUATE  CMA      <--  IR          << 5 >>
  6 EQUATE  CML      <--  IR          << 4 >>
  7 EQUATE  RAR      <--  IR          << 3 >>
  8 EQUATE  RAL      <--  IR          << 2 >>
  9 EQUATE  RT       <--  IR          << 1 >>
 10 EQUATE  IAC      <--  IR          << 0 >>
 11
 12
 13
 14
 15                                                          -->


Screen # 28
  0 ( bit names continued )
  1
  2 EQUATE  GP2-3    <--  IR          << 8 >>
  3 EQUATE  CLA      <--  IR          << 7 >>
  4 EQUATE  SMA      <--  IR          << 6 >>
  5 EQUATE  SZA      <--  IR          << 5 >>
  6 EQUATE  SNL      <--  IR          << 4 >>
  7 EQUATE  IS       <--  IR          << 3 >>
  8 EQUATE  OSR      <--  IR          << 2 >>
  9 EQUATE  HLT      <--  IR          << 1 >>
 10 EQUATE  GP-3     <--  IR          << 0 >>
 11
 12 EQUATE  SPA      <--  IR          << 6 >>
 13 EQUATE  SNA      <--  IR          << 5 >>
 14 EQUATE  SZL      <--  IR          << 4 >>
 15                                                          -->


Screen # 29
  0 ( bit names continued )
  1
  2 EQUATE  CARRY       <--  L           <<  0 >>
  3 EQUATE  C-IN        <--  C.HOLD      <<  0 >>
  4 EQUATE  NEG.AC      <--  AC          << 11 >>
  5 EQUATE  INTERRUPT   <-- STOP.FLAG    <<  0 >>
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15                                                          -->
```

```
Screen # 30
  0 ( mapping description for code generator )
  1
  2 45 MWORD.SIZE
  3
  4 FIELD ADDRF    <<  0 ::  7 >>
  5 FIELD ALUF     <<  8 :: 11 >>
  6 FIELD TSTF     << 12 :: 15 >>
  7 FIELD EOLF     << 16 :: 18 >>
  8
  9 ENCODE.ALU.SEL ALU1 INTO ALUF
 10   F=0 F=1 F=A F=B F=-A F=-B F=A+1 F=B+1 F=A+B F=A-B
 11   F=AVB F=A^B F=NOT.A F=NOT.B   END.LIST
 12
 13
 14
 15                                                              -->

Screen # 31
  0 ( mapping for enable statements )
  1
  2 AT ENB D1          SET BIT 20
  3 AT ENB D2          SET BIT 21
  4 AT ENB D3          SET BIT 22
  5 AT ENB D4          SET BIT 23
  6 AT ENB D5          SET BIT 24
  7 AT ENB D6          SET BIT 25
  8 AT ENB D7          SET BIT 26
  9 AT ENB D8          SET BIT 27
 10
 11 AT ENB MA          SET BIT 28
 12 AT ENB RAM         SET BIT 29
 13 AT ENB ALU1        SET BIT 30
 14 AT ENB MAP         SET BIT 31
 15 AT ENB SWITCHES    SET BIT 32                                -->

Screen # 32
  0 ( mapping for load statements )
  1
  2 AT LD AC           SET BIT 33
  3 AT LD L            SET BIT 34
  4 AT LD PC           SET BIT 35
  5 AT LD IR           SET BIT 36
  6 AT LD MA           SET BIT 37
  7 AT LD STOP.FLAG    SET BIT 38
  8 AT LD SWITCHES     SET BIT 39
  9 AT LD RAM          SET BIT 40
 10 AT LD C.HOLD       SET BIT 41
 11 ( mapping for test statements )
 12
 13 AT TST CARRY       SET TSTF TO 10
 14 AT TST C-IN        SET TSTF TO 11
 15 AT TST NEG.AC      SET TSTF TO 12                            -->
```

```
Screen # 33
  0 ( test statements continued )
  1
  2 AT TST IAC        SET TSTF TO 1
  3 AT TST RT         SET TSTF TO 2
  4 AT TST RAL        SET TSTF TO 3
  5 AT TST RAR        SET TSTF TO 4
  6 AT TST CML        SET TSTF TO 5
  7 AT TST CMA        SET TSTF TO 6
  8 AT TST CLL        SET TSTF TO 7
  9 AT TST PB         SET TSTF TO 8
 10 AT TST IB         SET TSTF TO 9
 11 AT TST GP-3       SET TSTF TO 1
 12 AT TST HLT        SET TSTF TO 2
 13 AT TST OSR        SET TSTF TO 3
 14 AT TST IS         SET TSTF TO 4
 15 AT TST SNL        SET TSTF TO 5                                    -->


Screen # 34
  0 ( test statements continued )
  1
  2 AT TST SZA          SET TSTF TO 6
  3 AT TST SMA          SET TSTF TO 7
  4 AT TST CLA          SET TSTF TO 8
  5 AT TST GP2-3        SET TSTF TO 9
  6 AT TST SZL          SET TSTF TO 5
  7 AT TST SNA          SET TSTF TO 6
  8 AT TST SPA          SET TSTF TO 7
  9
 10 AT TST INTERRUPT    SET BIT 44
 11
 12 AT TST AUTO.INC     SET TSTF TO 13
 13 AT TST IOT.OPR      SET TSTF TO 14
 14 AT TST ZERO.AC      SET TSTF TO 15
 15 AT TST USKIP        SET TSTF TO 00                                   -->


Screen # 35
  0 ( mapping for input/output subroutines )
  1
  2 AT CODE.GROUP GET.IN     SET BIT 43
  3 AT CODE.GROUP PUT.OUT    SET BIT 44
  4
  5 ( mapping for branch control words )
  6
  7 AT JVCTR          SET BIT 19
  8 AT JMP            SET EOLF TO 1   SET ADDRF TO LBL.VALUE
  9 AT JTR            SET EOLF TO 2   SET ADDRF TO LBL.VALUE
 10 AT JFS            SET EOLF TO 3   SET ADDRF TO LBL.VALUE
 11 AT JSR            SET EOLF TO 4   SET ADDRF TO LBL.VALUE
 12 AT RTN            SET EOLF TO 5
 13 AT RTN.TR         SET EOLF TO 6
 14 AT RTN.FS         SET EOLF TO 7
 15                                                                    -->
```

```
Screen # 36
  0 ( declare labels used in pseudo-microprogram )
  1
  2 38       LABELS
  3
  4 GP3               FETCH           1STEP           INC.PC
  5 IND.TST           INDIRECT        IND.END         LOGIC.AND
  6 TAD               ISZ             DCA             JMS
  7 JUMP              IOT             OUT.DATA        OPR
  8 CMAC              CMLINK          INCAC           ROR1
  9 ROR2              TST.CLA         TST.CMA         TST.CML
 10 TST.IAC           TST.RAR         TST.RAL         TST.2/3
 11 TST.SZA           TST.SMA         SKIP1           INVERT
 12 TST.SNA           TST.SPA         TST.USKIP       END.GP2
 13 TST.HLT           THE.END
 14
 15                                                              -->

Screen # 37
  0 ( I/O subroutines for pseudo-microprpgram )
  1
  2 : READ      R> DUP 2+ >R @ >BODY 2@ ;
  3 : WTREG     R> DUP 2+ >R @ >BODY 2! ;
  4
  5 CODE.GROUP : PUT.OUT  READ AC DROP EMIT ;
  6 CODE.GROUP : GET.IN   ?TERMINAL IF KEY ELSE 0 THEN 0 WTREG AC
  7
  8 ( pseudo-microprogram )
  9
 10 : PDP-8
 11
 12 [ GP3 ]     JSR TST.CLA
 13
 14 [ FETCH ]   TST INTERRUPT  JTR THE.END
 15                                                              -->

Screen # 38
  0 ( instruction fetch, decode and address calculation )
  1
  2 [ 1STEP ]  ENB D5  ENB RAM  TST IOT.OPR LD IR JTR INC.PC
  3            ENB D8  ENB D4   TST PB  LD MA JTR IND.TST
  4            ENB D8  ENB D3   TST IB  LD MA JTR INDIRECT
  5
  6 [ INC.PC ]
  7   ENB D4  ENB MAP SEL ALU1 F=A+1 ENB ALU1  LD PC  JVCTR
  8
  9 [ IND.TST ]   TST IB  JFS INC.PC
 10 [ INDIRECT ]  ENB MA  TST AUTO.INC  JFS IND.END
 11   ENB MA  ENB RAM  SEL ALU1 F=A+1 ENB ALU1  LD RAM  CONT
 12 [ IND.END ]  ENB MA  ENB RAM  ENB D2  LD MA  JMP  INC.PC
 13                                                              -->
 14
 15
```

```
Screen # 39
  0 ( instruction execution procedures )
  1
  2 [ LOGIC.AND ]
  3   ENB MA  ENB RAM  SEL ALU1 F=A#B ENB ALU1  LD AC  JMP FETCH
  4
  5 [ TAD ]
  6   ENB MA ENB RAM SEL ALU1 F=A+B ENB ALU1 LD AC LD L JMP FETCH
  7
  8 [ ISZ ]
  9   ENB MA ENB RAM SEL ALU1 F=A+1 ENB ALU1 LD RAM LD C.HOLD CONT
 10          TST C-IN  JFS FETCH
 11   ENB D4  SEL ALU1 F=A+1 ENB ALU1  LD PC  JMP FETCH
 12
 13
 14                                                              -->
 15


Screen # 40
  0 ( execution procedures continued )
  1
  2 [ DCA ]
  3   ENB MA  SEL ALU1 F=B ENB ALU1  LD RAM  CONT
  4   SEL ALU1 F=0  ENB ALU1  LD AC  JMP FETCH
  5
  6 [ JMS ]
  7   ENB MA  ENB D4  SEL ALU1 F=A  ENB ALU1  LD RAM  CONT
  8   ENB MA  ENB D1  SEL ALU1 F=A+1  ENB ALU1  LD PC  JMP FETCH
  9
 10 [ JUMP ]
 11   ENB MA  ENB D1  SEL ALU1 F=A  ENB ALU1  LD PC  JMP FETCH
 12
 13 [ IOT ]         TST IB  JTR OUT.DATA
 14                 PUT.OUT  JMP FETCH
 15 [ OUT.DATA ]    GET.IN   JMP FETCH                          -->


Screen # 41
  0 ( operate group instructions )
  1
  2 [ OPR ]      TST GP2-3  JTR TST.2/3
  3              JSR TST.CLA
  4              TST CLL  JFS TST.CMA
  5              SEL ALU1 F=0  TST CMA  ENB ALU1   LD L  JFS TST.CM
  6 [ CMAC ]     SEL ALU1 F=NOT.B  TST CML  ENB ALU1    LD AC
  7                       JFS TST.IAC
  8 [ CMLINK ]   SEL ALU1 F=NOT.B  TST IAC  ENB ALU1   LD L
  9                       JFS TST.RAR
 10 [ INCAC ]    SEL ALU1 F=B+1  TST RAR  ENB ALU1  LD AC  LD L
 11                       JFS TST.RAL
 12 [ ROR1 ]     TST RT  ENB D7  LD AC  LD L   JFS FETCH
 13 [ ROR2 ]     ENB D7  LD AC  LD L  JMP FETCH
 14
 15                                                              -->
```

```
Screen # 42
  0 ( operate group continued )
  1
  2 [ TST.CLA ]  TST CLA  RTN.FS
  3              SEL ALU1 F=0  ENB ALU1  LD AC  RTN
  4
  5 [ TST.CMA ]  TST CMA  JTR CMAC
  6 [ TST.CML ]  TST CML  JTR CMLINK
  7 [ TST.IAC ]  TST IAC  JTR INCAC
  8 [ TST.RAR ]  TST RAR  JTR ROR1
  9 [ TST.RAL ]  TST RAL  JFS FETCH
 10              TST RT  ENB D6  LD AC  LD L  JFS FETCH
 11              ENB D6  LD AC  LD L  JMP FETCH
 12
 13
 14
 15                                                          -->


Screen # 43
  0 ( operate continued )
  1
  2 [ TST.2/3 ]  TST GP-3  JTR GP3
  3              TST IS  JTR INVERT
  4              TST SNL  JFS TST.SZA
  5              TST CARRY  JTR SKIP1
  6 [ TST.SZA ]  TST SZA  JFS TST.SMA
  7              TST ZERO.AC  JTR SKIP1
  8
  9 [ TST.SMA ]  TST SMA  JFS END.GP2
 10              TST NEG.AC  JFS END.GP2
 11 [ SKIP1 ]
 12               ENB D4 SEL ALU1 F=A+1 ENB ALU1 LD PC JMP END.GP2
 13
 14
 15                                                          -->


Screen # 44
  0 ( operate continued )
  1
  2 [ INVERT ]      TST SZL  JFS TST.SNA
  3                 TST CARRY  JFS SKIP1
  4 [ TST.SNA ]     TST SNA  JFS TST.SPA
  5                 TST ZERO.AC  JFS SKIP1
  6 [ TST.SPA ]     TST SPA  JFS TST.USKIP
  7                 TST NEG.AC  JFS SKIP1
  8 [ TST.USKIP ]   TST USKIP  JTR SKIP1
  9 [ END.GP2 ]     JSR TST.CLA
 10                 TST OSR  JFS TST.HLT
 11                 TST HLT  ENB SWITCHES  SEL ALU1 F=AVB  ENB ALU1
 12                 LD AC  JFS FETCH
 13 [ TST.HLT ]     TST HLT  JFS FETCH
 14
 15 [ THE.END ]     ;                                        -->
```

```
Screen # 45
  0 ( FILL JUMP TABLE )
  1
  2 8 LOAD.VCTRS 0 MAP
  3          LOGIC.AND        TAD      ISZ      DCA
  4          JMS              JUMP     IOT      OPR
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
Screen # 46
  0  ( PDP-8 ASSEMBLER )
  1
  2 OCTAL
  3            VARIABLE LOCATION.POINTER$
  4 : =$       LOCATION.POINTER$ ! ;
  5 : OP$      CREATE , DOES> @ ;     ( DEFINING WORD USED FOR )
  6                                   ( OP-CODES WITH NO OPERAND )
  7 : OP1$     CREATE , DOES> @ [COMPILE] ' >BODY @
  8            177 AND OR 200 OR ;    ( OP-CODES WITH 1 OPERAND )
  9 : GROUP1$  CREATE , DOES> @ OR ;  ( MODIFIERS USED WITH )
 10                                   ( GROUP1 MICROINSTRUCTIONS )
 11 : SKIP$    CREATE 400 OR , DOES> @ OR ; ( SKIP GROUP )
 12 : LABEL$   CREATE 0 , DOES> LOCATION.POINTER$ @ SWAP ! ;
 13 : ;$       LOCATION.POINTER$ @ U.
 14            LOCATION.POINTER$ @ 2 * RAM MEM.FLD + OVER U. CR !
 15            1 LOCATION.POINTER$ +! ;                          -->
```

```
Screen # 47
  0  ( PDP-8 OPCODES )
  1
  2 OCTAL
  3
  4 0000     OP1$     AND$
  5 1000     OP1$     TAD$
  6 2000     OP1$     ISZ$
  7 3000     OP1$     DCA$
  8 4000     OP1$     JMS$
  9 5000     OP1$     JMP$
 10 6000     OP$      OUT$
 11 6001     OP$      ION$
 12 6002     OP$      IOF$
 13 6400     OP$      INN$
 14 7000     OP$      OPR$
 15                                                        -->
```

```
Screen # 48
  0 ( PDP-8 GROUP1 )
  1
  2 OCTAL
  3
  4 200      GROUP1$          CLA$
  5 100      GROUP1$          CLL$
  6 40       GROUP1$          CMA$
  7 20       GROUP1$          CML$
  8 10       GROUP1$          RAR$
  9 4        GROUP1$          RAL$
 10 12       GROUP1$          RAR2$
 11 6        GROUP1$          RAL2$
 12 1        GROUP1$          IAC$
 13
 14
 15                                                          -->


Screen # 49
  0  ( PDP-8 SKIP GROUP )
  1
  2 OCTAL
  3
  4 100      SKIP$    SMA$
  5 110      SKIP$    SPA$
  6 40       SKIP$    SZA$
  7 50       SKIP$    SNA$
  8 20       SKIP$    SNL$
  9 30       SKIP$    SZL$
 10 4        SKIP$    OSR$
 11 2        SKIP$    HLT$
 12 0        SKIP$    NO.OP$
 13
 14 : I$         400 OR ;       ( INDIRECT BIT )
 15 : P.ZERO$   7577 AND ;                                   -->


Screen # 50
  0  ( PDP-8  AUTO INCREMENT REGISTERS )
  1
  2 OCTAL
  3
  4 10       CONSTANT         R0$
  5 11       CONSTANT         R1$
  6 12       CONSTANT         R2$
  7 13       CONSTANT         R3$
  8 14       CONSTANT         R4$
  9 15       CONSTANT         R5$
 10 16       CONSTANT         R6$
 11 17       CONSTANT         R7$
 12
 13
 14
 15                                                          -->
```

```
Screen # 51
  0  ( PDP-8 LABELS )
  1
  2 OCTAL
  3
  4 LABEL$ A$
  5 LABEL$ B$
  6 LABEL$ C$
  7 LABEL$ D$
  8 LABEL$ E$
  9 LABEL$ F$
 10 LABEL$ G$
 11 DECIMAL
 12
 13 0 =$
 14
 15 CR 52 LOAD CR 52 LOAD


Screen # 52
  0 ( test program )
  1
  2 OCTAL
  3 0 =$
  4 B$ 7766 ;$       C$ 0 ;$          F$ 0 ;$          G$ 121 ;$
  5 20 =$
  6 D$       OPR$     CLA$     ;$
  7          TAD$     B$       ;$
  8          DCA$     C$       ;$
  9 A$       INN$               ;$
 10          OPR$     SNA$     ;$
 11          JMP$     A$       ;$
 12          DCA$     F$       ;$
 13          TAD$     F$       ;$
 14          OPR$     CMA$     ;$
 15          TAD$     G$       ;$                              -->


Screen # 53
  0 ( test program continued )
  1
  2 OCTAL
  3          OPR$     IAC$     ;$
  4          OPR$     SNA$     ;$
  5          OPR$     HLT$     ;$
  6          OPR$     CLA$     ;$
  7          TAD$     F$       ;$
  8 E$       OUT$               ;$
  9          ISZ$     C$       ;$
 10          JMP$     E$       ;$
 11          JMP$     D$       ;$
 12
 13 20 0 WT PC  DECIMAL
 14
 15
```

```
Screen # 54
  0 55  START.SCREEN
  1
  2 GEN.CODE  PDP-8
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15


Screen # 55
  0 0000  000000040020
  1 0001  10000002003D
  2 0002  00102102E005
  3 0003  002008828006
  4 0004  002008429007
  5 0005  0008C0880600
  6 0006  000000039005
  7 0007  00001003D009
  8 0008  010070000600
  9 0009  002030210005
 10 000A  000270010B01
 11 000B  000670010801
 12 000C  030070000600
 13 000D  00000003B001
 14 000E  000840810601
 15 000F  010050000300


Screen # 56
  0 0010  000240010001
  1 0011  010050800200
  2 0012  000850110601
  3 0013  000850110201
  4 0014  000000029016
  5 0015  100000010001
  6 0016  080000010001
  7 0017  000000029029
  8 0018  000000040020
  9 0019  000000037022
 10 001A  000440036023
 11 001B  000240035D24
 12 001C  000440031D25
 13 001D  000640034726
 14 001E  000604032001
 15 001F  000604010001
```

```
Screen # 57
  0 0020  000000078000
  1 0021  000240050000
  2 0022  00000002601B
  3 0023  00000002501C
  4 0024  00000002101D
  5 0025  00000002401E
  6 0026  000000033001
  7 0027  000602032001
  8 0028  000602010001
  9 0029  000000021000
 10 002A  000000024032
 11 002B  00000003502D
 12 002C  00000002A031
 13 002D  00000003602F
 14 002E  00000002F031
 15 002F  000000037039


Screen # 58
  0 0030  00000003C039
  1 0031  000840810639
  2 0032  000000035034
  3 0033  00000003A031
  4 0034  000000036036
  5 0035  00000003F031
  6 0036  000000037038
  7 0037  00000003C031
  8 0038  000000020031
  9 0039  000000040020
 10 003A  00000003303C
 11 003B  000340032A01
 12 003C  000000032001
 13
 14
 15
```