Computer Science Department
114 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

MICRO/40 Assembler Primer

by

H. K. Berg and E. Dekel

Technical Report 78-9
July 1978

Cover design courtesy of Ruth and Jay Leavitt

MICRO/40 Assembler Primer

by
Helmut K. Berg and Eliezer Dekel
Department of Computer Science
University of Minnesota

Abstract

This report is a tutorial guide to the concept and use of the MICRO/40 assembler. It is intended to familiarize new users with the microassembler as the fundamental microprogram development aid in the PDP-11/40E microprogramming support system. The introduction of the MICRO/40 syntax in BNF and a commented list of all error messages make this report also usable as a reference for advanced users. Microprogramming techniques that scope with the effects of hardware idiosyncrasies on PDP-11/40E microprogramming are presented in the form of a tutorial. The report covers the basics needed for the day-to-day use of the microassembler, such as the microprogram input formats, the invokation of MICRO/40, and the interpretation of microassembler outputs. The use of MICRO/40 under the UNIX operating system is demonstrated by a commented protocol of a complete terminal session. This terminal session also illustrates the techniques for error corrections using the UNIX text editor.

1. Introduction

The microassembler described in this report is the language facility in our microprogramming laboratory that assists the user in writing PDP-11/40E microcode. MICRO/40 was developed at Carnegie-Mellon University, Department of Computer Science, to run as cross assembler on a PDP-10 computer [1]. The MICRO/40 version we refer to in this report is an improved PDP-11 version of the original microassembler that was written at the Technical University Berlin, Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Betriebssysteme. It runs on a PDP-11/40 under the UNIX operating system [2,3] and is written in the "C" programming language [4]. MICRO/40 interfaces to a microsimulator [5] and a microprogram test system [6,7] which, together with the microassembler, constitute the kernel of our PDP-11/40E microprogramming support system.

The PDP-11/40E is a standard PDP-11/40 computer that has been extended by the following hardware features which were developed at Carnegie-Mellon University [1,8]:

- 1K 80-bit words of random access (RAM) control store for storing user microprograms.
- 32 80-bit words of read-only (PROM) control store for bootstrapping microprograms.
- a 16-word stack for temporary data storage.
- a shift and mask unit and a carry control unit which extend the data manipulation capabilities of the basic PDP-11/40 processor.

The 3-River Computer Corporation offers these hardware accessories as a writable control store option (WCS 11/40) for the PDP-11/40. The PDP-11/40E has a horizontal microinstruction format that allows user microprograms access to all functional hardware units and data paths in the basic PDP-11/40 processor and in the WCS 11/40. The MICRO/40 assembler supports the construction of microinstructions from any combination of microoperations in this microinstruction format. The microprogramming features of the PDP-11/40E are described in [1,9].

In a typical microassembler, mnemonics are defined for each legal microoperation. Microinstructions are constructed by combining these menmonics in assembly code lines. For vertical microinstruction formats, only a single (relatively complex) microoperation can generally be defined in each micro-

instruction. Contrastingly, several (relatively primitive) microoperations can be combined into a single horizontal microinstruction. That is, horizontal microinstruction formats allow the specification of parallel actions (microoperations) to be performed by the machine hardware. In addition to mnemonic microoperation definitions, microassemblers usually provide for the use of mnemonic labels. Furthermore, the placement of microinstructions in the control store is usually supported by microassemblers, although microcode is generally not relocatable.

More sophisticated microassemblers may include macro facilities, in addition to the above features. Microassemblers for horizontal microinstruction formats may also automatically insert default values into microoperation fields which have not been filled by the microprogrammer. Furthermore, attempts have been made to make microassemblers partly machine-independent. These attempts make a single assembly language usable to generate microcode for several different microprogrammable machines. To this end, the assembly language syntax is parameterized and the hardware of the target machine is described at the assembler directive level. Finally, self-documenting assemblers have been developed that generate a commentary for each microinstruction written by the microprogrammer and thus, support microcode documentation.

The MICRO/40 assembler described in the following sections includes all the standard features of a microassembler for horizontal microinstructions. Additionally, a macro definition facility is provided. Furthermore, compound statements are included into MICRO/40 to support some of the idiosyncrasies of the PDP-11/40E. Microprogramming in the MICRO/40 assembly language is facilitated by the fact that the assembler transforms (register-transfer) assignment statements into appropriate microoperation field specifications, and the automatic insertion of default values into unspecified microoperation fields.

Microcode testing and debugging are integral parts of the microcode construction process and should not be delayed until after a microprogram is written. Therefore, microprogramming languages must support microcode testing and debugging. Analogous to the relationship between high-level programming languages and assembly languages for software, high-level microporgramming languages have many advantages over microcode assembly languages, with respect to microcode construction. For example, program development techniques, such as structured programming, become applicable to microprogramming through the use of machine-independent high-level microprogramming languages. Although structured, high-level microcode generation also greatly facilitates error location and correction, there exists no widely accepted high-level microprogramming language. This situation is primarily due to the fact that hard-

ware-dependent timing conditions (especially for asynchronous operations) and the inherent parallelism hinder the compiler construction for machines with horizontal microinstruction formats. The fundamental problem, in this respect, is the generation of optimized microcode, and the lack of realistic models of microinstruction and microoperation semantics which, by nature, are considerably more complex than corresponding models for software semantics. That is, in essence, the provision of high-level microprogramming tools is hindered by the necessity to exploit hardware characteristics to produce efficient microcode. Unoptimized microcode is generally unacceptable, because of the frequency with which microinstructions are executed.

As a result of the discussed difficulties in developing high-level microprogramming facilities, typical microprogramming support systems are still based on a microassembler and the associated loader and simulator software. Microassemblers leave the task of microcode optimization to the microprogrammer, as they do not offer any assistance in exploiting the inherent parallelism in microinstructions. Generally, a microassembler merely checks the legality of the combination of microoperations which the microprogrammer specified within a microinstruction. These syntax checks in a microassembler may be considered as a type of static microprogram testing and therefore, are usually restricted to the detection of static errors. Dynamic microprogram testing via microprogram execution requires the availability of an off-line (soft) test system, such as a microcode simulator, or of on-line (hard) test systems, such as an interactive debugger or special hardware accessories for the instrumentation of microprogram executions.

The error detection capabilities of the MICRO/40 assembler are enhanced by the WCS 11/40 microinstruction format [1,9] which makes certain conflicts detectable through microcode examination. The typical conflicts that are detected by the MICRO/40 assembler are the illegal use of data paths, registers, and functional hardware units. With respect to timing errors, the microassembler checks the specified microoperations against the length of the specified processor clock cycle, and attempts to enforce the proper timing for each microinstruction. Nevertheless, the detection of timing conflicts by the MICRO/40 assembler is restricted to static errors.

MICRO/40 supports microcode testing and debugging by generating files to be used by a microcode simulator [5]. The microsimulator allows the investigation of effects of simulated microinstruction executions. The facilities for detecting and locating of errors are basically restricted to static errors which can be observed by investigating processor registers, including the microprogram pointer. For the detection of dynamic, hardware-dependent micro-

program errors, an on-line test system, SMILE [6,7], for microprogram load and examination is available in our microprogramming support system. The SMILE system loads the binary object microcode as supplied by MICRO/40 into the writable control store, and allows microprogram testing at the microprogram level by executing PDP-11/40 machine language instructions which call upon the execution of the loaded microcode. However, using the SMILE system, erroneous microoperations usually cannot be located by tests at the microprogram level. Therefore, the microprogramming support system has been complemented with a logic state analyzer whose use as an on-line test facility at the microinstruction and microoperation level is documented in [10].

This report is intended to provide a tutorial guide to the concept and use of MICRO/40 that expands and complements the available documentation [1,8]. To this end, we first introduce the MICRO/40 syntax in section 2. Based on the content of section 2, we proceed, in section 3, with a discussion of the conceptual features of MICRO/40 and associated programming techniques. Guidelines for the use of MICRO/40 and the interpretation of assembler output are presented in section 4. Finally, in section 5, we demonstrate a complete example of the microassembler operation in the form of a terminal session.

2. MICRO/40 Syntax

MICRO/40 has many of the limitations of a software assembler. Furthermore, being a microassembler for a specific machine (PDP-11/40E), register and microoperation field names as well as operator symbols are predeclared and cannot be redefined. The nature of register-transfer operations in the PDP-11/40E processor as described by MICRO/40 assembly language lines implies that all names and operator symbols are global.

2.1 Input Format

Each line of MICRO/40 input is assembled into a single 80-bit microinstruction (for exceptions to this rule, cf. subsection 2.5). A line of MICRO/40 input is defined as all the characters between two consecutive line-feeds. Any line ending with a hyphen(-) has the next line concatenated to it, i.e., the hyphen is the line continuation symbol. Each line is read and processed individually, and no distinction is made between lower case and upper case letters. The MICRO/40 assembly language is a free-format language in which statements can be placed anywhere on the line with as many blanks as needed to make the described microinstruction legible. Anything following an exclamation point(!) on a line is considered a comment and is ignored.

2.1.1 Identifiers

A MICRO/40 identifier is a character string starting with an alphabetic (A - Z, a-z) and followed by an arbitrary number of alphabetics (A - Z, a-z), digits (0-9), a dot(.), a slash(/), or an underline(_). The number of significant identifier characters is limited to 20.

2.1.2 Numbers

A numeric constant is a sequence of one or more digits. All numeric constants are considered to be octal, unless the number contains the digits 8 or 9, or a decimal point in which case, it is interpreted as a decimal number. The syntax of a negative number is, -<number>. Negative numbers may be parenthesized. The above rules define the following equivalences, e.g.,

137 is equivalent to 95,
102 is equivalent to 66. ,
-32 is equivalent to (-32).

2.1.3 Labels

A label declaration is an identifier followed by a colon(:), i.e.,

<label declaration>::=<label>:
<label>::=<identifier> ,

A line may contain an arbitrary number of labels which may be placed anywhere on the line.

2.2 Microoperation Field Assignments

The generic use of the MICRO/40 assembler is to assign specific values to each microoperation field in the horizontal microinstruction. The syntax[1] for a field assignment is:

<field name> = <value> {;} .

The semi-colon (;) separates multiple field assignments on a single MICRO/40 input line. The semi-colon following the last field assignment on a line may be omitted.

---

1) Throughout this text, we use the meta symbol {}to enclose optional objects in syntax definitions (i.e., 0 or 1 repetitions of the enclosed object are allowed).

2.2.1 Microoperation Fields

The format of the 80-bit PDP-11/40E microinstruction (XU) and the bit position assignment of the microoperation fields are shown in Fig. 1. The 36 microoperation field in XU may be divided into 14 groups [9]. The

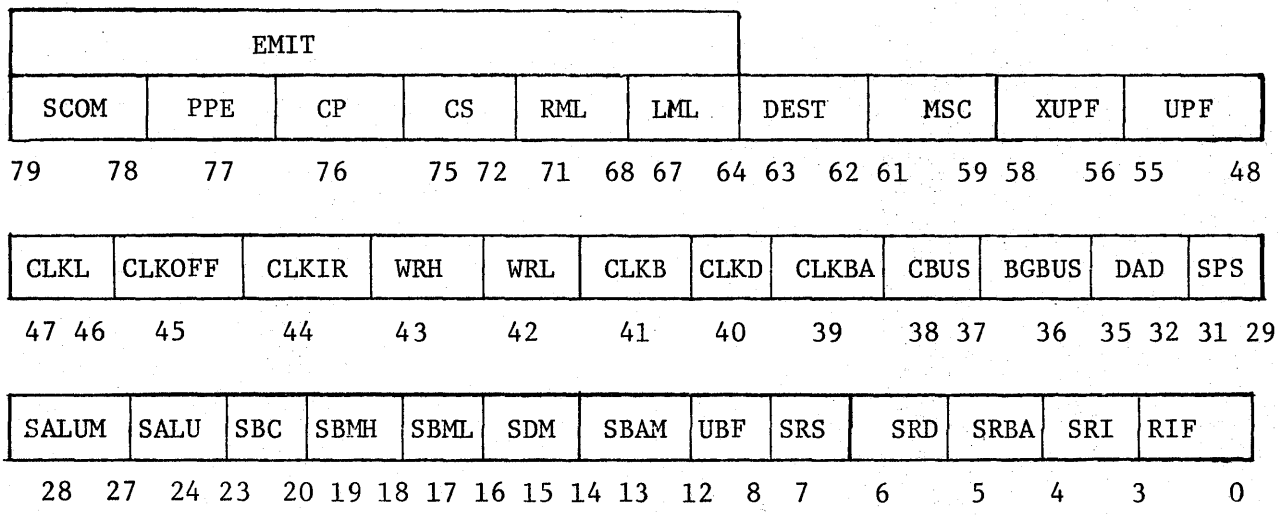


Figure 1: XU Format

PDP-11/40E registers, data paths and functional units associated with the microoperations are depicted in the register-transfer block diagrams of the PDP-11/40 processor, the WCS 11/40, and their mutual interfaces as given in the appendix.

Clock Control

CLKL (XU<47:46>): Processor Clock Length Control
Allows a selection from three basic PDP-11/40 processor cycle times.

CLKOFF (XU<45>): Processor Clock Off
When set, turns processor clock off.

Register Load Control

CLKIR (XU<44>): Clock Instruction Register
Allows clocking the D MUX output into the instruction register.

WRH (XU<43>): Write High Order Byte of DMUX BUS
Allows writing the high order byte of the D MUX output into a selected general purpose register.

WRL (XU<42>): Write Low Order Byte of DMUX BUS
Allows writing the low order byte of the D MUX output into a selected general purpose register.

CLKB (XU<41>): Clock B Register
Allows clocking the D MUX output into the B Register.

CLKD (XU<40>): Clock D Register
Allows clocking the ALU output into the D Register.

CLKBA (XU<39>): Clock BA Register
Allows clocking the BA MUX output into the BA Register

UNIBUS Control

CBUS (XU<38:37>): Control of UNIBUS
Allows the specification of UNIBUS data transfers.

BGBUS (XU<36>): Begin a UNIBUS Transfer
Allows the initiation of a UNIBUS data transfer specified in CBUS.

Instruction Processor Logic Control

DAD (XU<35:32>): Discrete Alteration of Data
Directs the ALU control logic (which is associated with the instruction decoding logic) as to alterations of operations to be performed by functional hardware units in the data processor.

Processor Status Control

SPS (XU<31:29>): Select Processor Status
Determines loading of the PS Register from the DMUX BUS, clocking of condition codes into the PS Register, and gating of the PS Register onto the RD BUS.

ALU Control

SALUM (XU<28>): Select ALU Mode
Selects ALU mode of operation (arithmetic or logic)

SALU (XU<27:24>): Select ALU Function
Allows the selection of up to 16 arithmetic or 16 logic ALU functions.

Multiplexor Control

SBC (XU<23:20>): Select Input to B MUX from B Constants
Allows the selection of a constant to be gated to the ALU B-input.

SBMH (XU<19:18>): Select Input to B MUX's High Order Byte
Allows the selection of bytes from the B Register and the B Constants to be gated to the high order byte of the ALU B-input.

SBML (XU<17:16>): Select Input to B MUX's Low Order Byte
Allows the selection of bytes from the B Register and the B Constants to be gated to the low order byte of the ALU B-input.

SDM (XU<15:14>): Select Input to D MUX
Select the RD BUS, the UNIBUS data lines, the D Register, or the right-shifted D Register as the input to D MUX.

SBAM (XU<13>): Select Input to BA MUX
Selects the output of the ALU or the RD BUS as the input to BA MUX.

General Purpose Register Addressing Control

SRS (XU<7>): Select General Purpose Register Address from IR Source Field
Allows IR <8:6> to be used as a source of a general purpose register address.

SRD (XU<6>): Select General Purpose Register Address from IR Destination Field
Allows IR <2:0> to be used as a source of a general purpose register address.

SRBA (XU<5>): Select General Purpose Register Address from the BA Register
Allows BA <3:0> to be used as a source of a general purpose register address.

SRI (XU<4>): Select General Purpose Register Address from RIF
Allows XU <3:0> = RIF to be used as a source of a general purpose register address.

RIF (XU<3:0>): Register Immediate Field
Used as source of a general purpose register address when enabled by SRI.

Microinstruction Sequencing Control

UBF (XU<12:8>): Micro Branch Field
Specifies the branch micro test (BUT) to be performed, in order to generate the address of the successor microinstruction by ORing the determined basic microbranch code (BUBC) into UPP<5:0>.

UPF (XU<55:48>): Microprogram Pointer Field
Used to specify the address of the next microinstruction to be executed. The specified address may be modified as a result of a branch micro test (BUT) specfied in UBF.

XUPF (XU<58:56>): Extended Microprogram Pointer Field

Concatenated with UPF (XU<55:48>), this field froms an 11-bit microinstruction address in the extended control store address space (addition of the RAM and PROM control stores to the ROM control store).

WCS 11/40 Data Paths Control

MSC (XU<61:59>): Mask/Shift Control

DEST (XU<63:62>): Destination

The DEST and MSC fields are combined into a 5-bit field that specifies how the bits XU<79:64> are to be interpreted (as function fields or EMIT field) and how the WCS 11/40 data paths are set up for the interpretation of the current microinstruction.

Mask/Shift Control

LML (XU<67:64>): Left Mask Limit

Specifies the number of bits of the S MUX output that are to be masked off from the left.

RML (XU<71:68>): Right Mask Limit

Specifies the number of bits of the S MUX output that are to be masked off from the right.

SC (XU<75:72>): Shift Count

Specifies the number of bit positions (between 0 and 15) for a right rotate of the S MUX output

Depending on the specification of DEST/MSC, this field might also be used to specify a 4-bit value to be transferred into the stack pointer SP.

Carry Control

CP (XU<76>): Carry Propagate Control

Specifies the application of the content of CPFF (Carry Propagate Flip-Flop) to the carry input of the ALU and the storage of a new carry bit (as determined by SCOM) into CPFF.

SCOM (XU<79:78>): Select Carry Out Multiplexer

Specifies the selection of a carry bit from the ALU carry-out bits (ALU15, word carry, byte carry) or the condition code bit PS(C) of the processor status word to be stored into the high-order bit extension of the D Register (D(C)) and into CPFF.

Stack Control

PPE (XU<77>): Push/Pop Enable

Specifies, if stack read/write operations are combined with pop/push operations, respectively.

Arithmetic and Addressing Constants

EMIT (XU<79:64>): Microliteral Field

Used to specify 16-bit arithmetic or addressing constants. The use of XU<79:64> for microliterals is determined by the specification in DEST/MSC. If XU<79:64> is used as a microliteral field, the fields LML, RML, SC, CP, PPE, SCOM of the XU WORD serve as a data register whose content is transferred to the input of S MUX.

2.2.2 Use of Field Assignments

With each field assignment in a MICRO/40 assembly language program, the appropriate microoperation field is cleared and the assigned value is stored in it. If a single MICRO/40 input line contains two different assignments to the same field, only the last (right-most in the lint) value is considered and a warning is issued. That is, the microassembler does not prevent double field assignment on the same line. All values assigned to microoperation fields are expected to be assembly time constants (numbers), except for values assigned to the EMIT and XUPF fields.

The EMIT field may store a 11-bit control store address that is associated with a mnemonic label in the assembly microprogram, or the 16-bit address of a 16-bit field in the writable control store. The latter option allows the writable control store (RAM) to be accessed as a data scratch pad (cf. subsections 2.5.7 and 2.5.8).

MICRO/40 uses the name XUPF to denote the extended microprogram pointer field, XU<58:48> = XUPF, UPF (cf. subsection 2.2.1). This field always holds the base address of the next microinstruction, as there is no microinstruction counter in the PDP-11/40E. If no assignment is made to the XUPF field, the increment of the current microinstruction (control store) address is automatically assigned as a default. This assignment corresponds to a 'goto next consecutive microinstruction'. Assigning a constant to XUPF causes a 'goto' to that absolute control store location. However, mnemonic labels may also be assigned

to XUPF, in which case, the associated 11-bit microinstruction address is inserted into XU<58:48>.

For MICRO/40 field assignments, some of the microoperation fields discussed in the preceding subsection are concatenated into compound fields:

```
ALU = SALUM,SALU
BUS = CBUS,BGBUS
CLK = CLKL,CLKOFF
SBM = SBMH,SBML
SRX = SRS,SRD,SRBA,SRI
 WR = WRH,WRL
XUPF = XUPF,UPF    .
```

The microinstruction generated by the following field assignments transfers the constant 10 into the B Register.

```
emit = 10; msc = 1; clk = 6; clkb = 1
```

The assembler sets all other microoperation fields to 0, except for the XUPF field which is assigned the control store address of the next consecutive microinstruction, if there exists one. The assignment of the binary value, 00 001, to the DEST/MSC fields specifies the data transfer, RD ← EMIT. The assignment clkb=1 opens the data path, B ← DMUX, while the implicit setting sdm=0 defines the transfer DMUX ← RD BUS. Hence, we have the compound transfer:

B ← DMUX ← RD BUS ← EMIT = 10 .

The assignment clk=6, i.e., CLKL=3, selects a P3 processor clock cycle for this data transfer (cf.[9]). Generally, MICRO/40 automatically assigns an appropriate clock cycle length for each microinstruction. Thus, the field assignment, clk=6, is redundant.

To further illustrate the use of field assignments in the MICRO/40 assembly language we give the following example microprogram for adding the numbers from 1 to 10 that stores the result in the general purpose register R[0]. The verification of this microprogram is left to the reader (cf.[1],[9]).

```
    wr=3; clkd=1; alu=23; sdm=2; srx=1
        ! R[0]=0
    emit=10; msc=1; clkb=1
        ! B← 10
loop:emit=177777;msc=1;clkb=1;clkd=1;alu=11;sdm=2;ubf=12;xupf=n
        !D←B-1;B←D;goto n;skip the microinstruction following -
        !the microinstruction with label n, if D=0
    xupf = next
        ! goto next
```

```
n:wr=3;clkd=1;alu=11;sdm=2;srx=1;xupf=loop
    ! R[o] ← R[o] + B; goto loop
next: <whatever comes next>
```

2.3 Assignment Statements

As discussed in the preceding subsection, microprograms may be written in the form of field assignments. However, the exclusive use of field assignments makes microprogramming a tedious task. Therefore, MICRO/40 provides for the specification of carrier-to-carrier transfers in the form of assignment statements. A carrier is a facility for accommodating bit strings which represent the information in a computer. A carrier may be a register, a memory, or a data paths. The microoperation fields associated with a carrier-to-carrier transfer are automatically set to appropriate values, when an assignment statement is assembled.

2.3.1 Carrier Identification

R: General Purpose Registers

The PDP-11/40 has 16 16-bit general purpose registers which are implemented as a data scratch pad. They are loaded from the DMUX BUX, supply output to the RD BUS, and are addressed by 4-bit addresses. In each microinstruction, only a single general purpose register can be addressed, i.e., simultaneous register read and write operations are not permissible.

The syntax for general purpose registers is:

```
<general purpose register>::=R[<index>]{<<selector>>}
<index>::= 0|1|2|3|4|5|6|7|10|11|12|13|14|15|16|17|8.|9.|10.|
           11.|12.|13.|14.|15.|BA|DF|SF
<selector>::= 1|h    .
```

A number directly specifies one of the 16 general purpose registers. Registers are specified indirectly by using the specifiers BA, DF, or SF. Using the specifier BA, BA<3:0> is taken as the source of the register address, whereas the specifiers DF and SF specify the destination field, IR<2:0>, and the source field, IR<8:6>, of the instruction register as the source of the register address, respectively. Furthermore, the optional selector allows the selection of either the low-order byte (1), R<7:0>, or the high-order byte (h), R<15:8>, of a general purpose register R.

D: Data Register

The D Register is a 16-bit register for the temporary storage of ALU output data.

Syntax: <D Register>::=d

DSHIFT: Right-Shifted D Register

The D Register is right-shifted at D MUX, such that DSHIFT = D(C), D<15:1>. D(C) is a 1-bit extension of the D Register that may be loaded with ALU carry outputs (ALU15, word carry, byte carry) or the condition code bit, PS(C), of the processor status word. DSHIFT can only appear on the right hand side of an assignment statement.

Syntax:<Right-Shifted D Register>::=dshift|d/2

B: ALU B-input Register

The B Register is a 16-bit register for the temporary storage of ALU B-input data. The syntax for the B Register is:

<B Register>::= b{<<B modifier>>}
<B modifier>::= <high selector><low selector>
<high selector>::= H|E|L|C
<low selector>::= H|Z|L|C .

The optional B modifier can only be used, if the B Register appears on the right hand side of an assignment statement. The high selectors select the following gating mechanisms:

H: BMUX<15:8> ← B<15:8>
E: BMUX<15:8> ← B<7>
L: BMUX<15:8> ← B<7:0>
C: BMUX<15:8> ← B CONSTANT<15:8>

The low selectors select the following gating mechanisms:

H: BMUX<7:0> ← B<15:8>
Z: BMUX<7:0> ← B<7:0>
L: BMUX<7:0> ← B<7:0>
C BMUX<7:0> ← B CONSTANT<7:0>

If no B modifier is specified, the default b<HZ> is assumed.

C: ALU B-input Constants

The C Register is a combinatorial network that provides basic constants to be used in processor operations (cf. [1,9]). The syntax for the Constant Register is:

<Constant Register>::=c[<number>]
<number>::= 0|1|2|3|4|5|6|7|10|11|12|13|14|15|16|17|
8.|9.|10.|11.|12.|13.|14.|15.

BA: UNIBUS Address Register

The BA Register is a 16-bit register for the temporary storage of UNIBUS addresses. It is decoded to detect processor register addresses in the UNIBUS addressing scheme. The BA Register can only occur on the left hand side of an assignment statement.

Syntax:<BA Register>::= ba

IR: Instrustion Register

The instruction register is a 16-bit register that holds PDP-11/40 machine language instructions. Its output is applied to the instruction decoding logic, is used to control microbranching, and specifies general purpose register addresses. The instruction register can only appear on the left hand side of an assignment statement.

Syntax: <Instruction Register>::= ir

PS: Processor Status Register

The processor status register is a 16-bit register which holds the processor status word that specifies condition codes, processor priority, trap condition, and operational modes.

Syntax: <Processor Status Register>::= ps

S: Stack

The WCS 11/40 stack is a 16 16-bit word register-memory that can be used as a data or address push-pop stack. It is addressed from the stack pointer (SP), receives input from E MUX, and can supply output to S MUX and UPP MUX.

Syntax: <stack>::= s

The use of the identifier s in a MICRO/40 line implies that the WCS 11/40 stack is used as a push-pop stack.

TOS: Top of Stack

TOS refers to the current top of the WCS 11/40 stack. However, references to TOS do not push/pop the stack.

Syntax: <Top of Stack>::= tos

SP: Stack Pointer

The stack pointer is a 4-bit register that holds the stack address. It can be set from XU<75:72> = SC<3:0>, and is incremented or decremented for pop or push operations, respectively.

Syntax: <Stack Pointer>::= sp

UNIBUS

In MICRO/40, the UNIBUS data lines can only be used as a source (right hand side of an assignment statement) that provides input data to registers on the DMUX BUS.

Syntax: <UNIBUS>::= unibus

EUBC: Extended Microbranch Condition BUS

The EUBC BUS supplies 11-bit microinstruction control store addresses to the XUPP section of the microinstruction buffer. The EUBC BUS can only occur on the left hand side of an assignment statement. Input to the EUBC BUS may be supplied from the stack and from the EMIT field in the microinstruction.

Syntax: <EUBC BUS>::= eubc

RAM: Writable Control Store

The writable control store has a capacity of 5K 16-bit words. It receives 16-bit data inputs and supplies 80-bit words at the output. It can be used to store 1K of 80-bit microinstructions, to store 5K of 16-bit data words, or as a combination of control store and data scratch pad. The syntax for RAM references is:

<RAM reference>::= RAM[<specifier>]

<specifier>::= s|tos .

2.3.2 Syntax of Assignment Statements

The basic form of an assignment statement is:

<assignment>::= <carrier list>_<expression>

<carrier list>::=<left carrier>|<carrier list>,<left carrier>

<left carrier>::= <general purpose register>|d|s|tos|sp|b|
ba|ir|eubc|<RAM reference>|ps

<expression>::= <simple expression>|<compound expression>

A <left carrier> is any of the carriers that can occur on the left hand side of

an assignment statement. The assignment operator is the underline (_). Assignment statements within a single line are separated by a semi-colon(;). The semi-colon following the last statement in a line is optional.

A simple expression is a right hand side of an assignment statement in which only a single carrier is specified, i.e.,

```
<simple expression>::= <general purpose register>|ps|unibus|
                       d|dshift|d/2|<B Register>|<Constant
                       Register>|<extension operand>
<extension operand>::= <extension carrier>{<<field selection>>}
                       {<shift><number>}
<extension carrier>::= s|tos|sp|<RAM reference>|<emit field>
<emit field>::= <number>|<label>|<table reference>
<shift>::= shift|^        .
```

(for the definition of <table reference>,cf. subsection 2.5.7). All numbers given in an extension operand specification are interpreted as decimal numbers, despite of their representation. The field selection defines a masking by specifying the right-most and the left-most bit position to be masked out of the specified extension carrier. The result of a mask operation is always right-adjusted at the output of the shift/mask unit in the WCS 11/40. The shift is a left shift by the specified number of bit positions. The combination of the shift and the right/left mask allows the extraction of any contiguous n-bit field ($1 \leq n \leq 16$) from an extension carrier.

A compound expression includes an ALU operation. The syntax for the ALU operators is:

```
<negation>::= not|~|\5|\32
<disjunction>::= or|!|\37
<conjunction>::= and|&|\4
<exclusive-or>::=xor|\26
<addition>::= plus|+
<subtraction>::= minus|-     .
```

To define an operator by a code number requires that a backslash (\) is typed before the code number. The syntax of a compound expression is defined:

```
<compound expression>::= <A-op> or <B-op>|<A-op> or not <B-op>|minus 1|
                         <A-op>|plus <A-op> and not <B-op>|
                         (<A-op> or <B-op>) plus <A-op> and not <B-op>|
                         <A-op> minus <B-op> minus 1|<A-op> and not <B-op> minus 1|
```

<A-op> plus <A-op> and <B-op>|<A-op> plus <B-op>|
(<A-op> or not <B-op>) plus <A-op> and <B-op>|
<A-op> and <B-op> minus 1|
<A-op> plus <A-op>|(<A-op> or <B-op>) plus <A-op>|
(<A-op> or not <B-op>) plus <A-op>|
<A-op> minus 1|<A-op> plus 1|(<A-op> or <B-op>) plus 1|0|
<A-op> plus <A-op> and not <B-op> plus 1|
(<A-op> or <B-op>) plus <A-op> and not <B-op> plus 1|
<A-op> minus <B-op>|<A-op> and not <B-op>|
<A-op> plus <A-op> and <B-op> plus 1|
(<A-op> or not <B-op>) plus <A-op> and <B-op> plus 1|
<A-op> and <B-op>|<A-op> plus <A-op> plus 1|
(<A-op> or <B-op>) plus <A-op> plus 1|
(<A-op> or not <B-op>) plus <A-op> plus 1|not <A-op>|
not (<A-op> or <B-op>)|
not <A-op> and <B-op>|not (<A-op> and <B-op>)|
not <B-op>|<A-op> xor <B-op>|
<A-op> and not <B-op>|not <A-op> or <B-op>|
not (<A-op> xor <B-op>)|
<A-op> and <B-op>|<A-op> or not <B-op>|
<A-op> or <B-op> .

<A-op> may be anything that can be placed on the RD BUS, i.e.,

<A-op>::=<general purpose register>|ps|<extension operand>

(for the definition of an extension operand, cf.<simple expression>). If multiple occurrences of <A-op> are needed, they should all be identically specified in the compound expression. The syntax for <B-op> is defined:

<B-op>::=<B Register>|<Constant Register> .

To demonstrate the use of MICRO/40 assignment statements, we rewrite the microprogram for adding the numbers from 1 to 10 (cf. subsection 2.2.2).

```
        d_0; R[0] _d
             !clear our accumulator R[0]
        b_10
             !clock the value 10 from the EMIT field into the B Register
   loop: d_177777+b; b_d; ubf=12; xupf=n
             !decrement the B Register using the constant -1 from the
             !EMIT field, break the loop if D=0.
```

```
      xupf=next
           !goto next
   n: d_R[0]+b; R[0]_d; xupf=loop
           !add the content of the B Register to R[0].
   next:  <whatever comes next>
```

Note that the use of assignment statements does not reduce the number of code lines, as each MICRO/40 line is assembled into a single microinstrucion.

2.3.3 Semantics of Assignment Statements

When writing assignment statements, it must be remembered that MICRO/40 assignments represent carrier-to-carrier transfers. That is, each transfer must be described explicitly. Hence, an assignment statement

```
b_177777+b
```

is not permissible. To decrement (add -1) the content of the B Register requires that the constant 177777 from the EMIT field is gated to the A-input of the ALU, the B Register is gated to the ALU B-input, and the ALU performs an addition. The result of this addition is transferred into the D Register. Then, the content of the D Register can be transferred, via D MUX, into the B Register. Therefore, it is necessary to write the assignment statements,

```
d_177777+b; b_d
```

to affect the two transfer operations. The assignment statements,

```
b_d; d-177777
```

are identical in their effect to writing them in reversed order, as both transfers are performed quasi parallel, i.e., within the same microinstruction execution cycle.

To further stress the point that MICRO/40 assignment statements correspond to carrier-to-carrier transfers, we consider the following example:

```
d_400; R[6]_d
```

The microprogrammer might hope that these assignment statements would set R[6] to 000400_8. MICRO/40 permits these assignment statements, however, their semantics deviate from the microprogrammers intention. As R[6] is referenced, its content is clocked onto the RD BUS. At the same time, the constant 400 in the EMIT field is clocked onto the RD BUS, and the contents of R[6] and EMIT are ORED. Hence, the new value of R[6] is its old value with bit 8 set ot 1. To perform the intended task, the assignment statments given above have to be executed in two consecutive microinstructions, i.e.,

```
d_400
R[6]_d
```

Note that, in general, gating two sources onto the same carrier causes an error. However, in the case of the RD BUS, any of the three potential sources (general purpose registers, processor status register, and extension) can idenpendently gate (OR) a word onto the bus.

Whenever the identifier, s, is used on the left hand side of an assignment statement, the stack is pushed before the value is stored. When tos is used, the new value is written over the current top of stack. On the right hand side of an assignment statement, s results in the value being read out of the stack and the stack being popped. Contrastingly, tos results in the value on the current top of stack being read out.

The semantics of WCS 11/40 shift/mask operations, which may be combined with any extension operand, are illustrated by the following example:

```
d_s<11:3>^2    .
```

The assignment statement places 9 bits (S<11:3>) of the 16-bit value popped off the stack into D<10:2>. That is, after bits 11 to 3 of S have been masked out, they are right-adjusted. Then, the resulting 16-bit word is rotated two positions to the left. The net effect is a 1-bit right shift of S<11:3>

```
      S    0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0
 S<11:3>   0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1  (Mask)
S<11:3>^2  0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0  (Shift)
```

2.4 Comments and Continuation Lines

Anything following an exclamation point (!) on a line is considered a comment and is ignored by the assembler. Any line that ends with a hyphen (-) has the next line concatenated to it. That is, two or more MICRO/40 lines may be concatenated into a single logical line to produce a single microinstruction after assembly. Lines may also be continued within a comment. Hence, the following six examples are equivalent.

```
1.  R[7]_d
2.  clk=2; rif=7; wr=3; srx=1; sdm=2
3.  clk=2; rif=7; wr=3; srx=1; sdm=2 ! D Register to R[7]
4.  rif=7; wr=3; -
    clk=2; srx=1; sdm=2 ! D Register to R[7]
5.  sdm=2; srx=1; clk=2;! P1 is a sufficient clock cycle time -
    rif=7; wr=3;
6.  rif=7; wr=3; srx=1;sdm=2 ! no need to specify clock cycle time
```

A microinstruction may be described by any number of concatenated lines, provided that the so constructed logical line does not exceed 300 characters.

2.5 Pseudo Operators

A number of pseudo operators are provided by MICRO/40 to make some of the idiosyncrasies of PDP-11/40E microprogramming transparent to the user and to further relieve the microprogrammer of the tedious task of constructing microprograms through explicit microoperation field assignments. Pseudo operators range from field assignments to compound statements that comprise several microinstructions. They facilitate processor clock control, structuring of microcode in the control store, microinstruction sequencing, and the use of the writable control store as a data scratch pad.

2.5.1 CLKOFF

MICRO/40 concatenates the clock length field and the clockoff field into a single 3-bit clock field, i.e., CLK=CLKL,CLKOFF. Hence, the CLKOFF bit can only be set, if, at the same time, a processor clock cycle length is specified by the microprogrammer. This restriction is due to the fact that all bits in a microoperation field are cleared, before a field value is stored. Furthermore, it is generally not possible to exploit the microassemblers capability to determine an appropriate clock cycle length and to automatically set the CLKL field, in microinstructions in which the CLKOFF bit is to be set. Therefore, the clkoff pseudo operator was implemented to overcome these difficulties.

The clkoff pseudo operator has the effect of turning on the CLKOFF bit in the CLK field. When the CLKOFF bit is set, the processor clock will be turned off at the end of the execution of the current microinstruction. The clkoff pseudo operator is independent of field assignments to the CLK field and may occur anywhere on a MICRO/40 line. That is, the microassembler implicity combines the clkoff pseudo operator with the specification of the processor clock cycle length.

2.5.2 NOOP

The noop pseudo operator is implemented to provide a means for generating a null (all microoperation fields set to o) microinstruction. This operator is needed, as a blank line is skipped by the microassembler and does not generate a microinstruction. The noop pseudo operator is to be used on a line by itself. It is ignored, if it occurs together with other MICRO/40 statements, such as field assignments or assignment statements.

2.5.3 FINIS

The finis pseudo operator indicates the end of the microcode source file. Any text following finis is ignored by MICRO/40. The finis pseudo operator is to be used on a line by itself. It does not generate an object microinstruction. If the microcode source file does not contain finis, the microassembler issues a warning and automatically inserts the finis pseudo operator into the source microcode (as the last line of the source file).

2.5.4 Dot(.)

The pseudo operator, dot(.), generally stands for the control store address of the microinstruction in which it occurs. There are two alternative applications of '.'.

The dot (.) may be used in field assignments (cf. subsection 2.2.2) to assign the control store address of the current microinstruction to the XUPF or EMIT fields. In these field assignments, the dot is used as follows:

```
                    emit = .
or                  xupf = .        .
```

Note that the address of the current microinstruction need not explicity be stated in the microinstruction, but is determined during assembling. MICRO/40 automatically assigns values to the xupf field, and thereby, may even re-arrange the microinstructions as given in the source microcode into a different ordering in the object code, in order to guarantee that the xupf field of a microinstruction that must proceed any particular microinstruction points to the correct control store location.

The second application of '.' is to force MICRO/40 to locate a micro-instruction at a particular control store location. To this end, an octal control store address is explicity assigned to the dot. The syntax for assignments to the dot is:

```
. = <value>    ,
```

where <value> must be in the range [2000_8: 3777_8] (RAM address space). The 'dot' assignment may occur anywhere on a MICRO/40 line.

2.5.5 LOWLIM

This pseudo operator provides a means to set the lowest control store address into which object microcode will be stored. The lowlim pseudo operator is used as follows:

```
lowlim = <value>    ,
```

where <value> must be in the range $[2000_8: 3777_8]$. If this pseudo operator does not occur in a microcode source file, lowlim is automatically set to the default value 2000. If used, lowlim = <value> should be the first MICRO/40 line in the microcode source file.

2.5.6 C.,N.Z.V.,N.Z.V.C.

These pseudo operators set the SPS (Select Processor Status) field in the microinstruction and enforce an appropriate processor clock cycle length in the CLK field. The condition codes N,Z,V, and C in the processor status word are defined as follows:

N=1, if the result of the last ALU operation was negative,
Z=1, if the result of the last ALU operation was zero,
V=1, if the last ALU operation resulted in an arithmetic overflow,
C=1, if the last ALU operation resulted in a carry from the most significant bit.

The pseudo operators c.,n.z.v., and n.z.v.c. set the SPS field to the values 1, 2, and 3, respectively, to cause the associated condition code bits in the processor status word to be set to the current conditions. Microinstructions which contain one of these pseudo operators must have a P1 or P3 processor clock cycle (cf. [1,9]). The pseudo operators c., n.z.v., and n.z.v.c. are mutually exclusive and may occur anywhere on a MICRO/40 line.

2.5.7 TABLE

The table pseudo operator facilitates the use of parts of the writable control store as a data scratch pad. Using explicit field assignments (cf. RAM references, subsection 2.2.2) the RAM may also be utilized as data scratch pad. However, it is difficult to explicity generate addresses of 16-bit RAM fields at run time. The table pseudo operator greatly supports the microprogrammer in this address generation by allowing for relative addressing within a declared data scratch pad. It allocates four 16-bit data words in an 80-bit control store location and treats a table as a zero origin array. Note that five 16-bit entries are actually available in an 80-bit control store location, but that it is difficult to generate the address of the fifth entry (cf. [1,9]).

The syntax for table declarations and references is:

<table declaration>::= table <table name> <size>

<table name>::= <identifier>

<size>::= <number>

<table reference>::= <table name>[<table index>],

where <table index> is a number in the range [0:<size>]. The following example illustrates the above syntax definition.

table INFO 15

INFO [0] refers to the first table entry.

INFO [1] refers to the second table entry.

:

INFO [14] refers to the fifteenth table entry.

The pseudo operator, table, would reserve 4 control store locations for the table, INFO, as 15_8 corresponds to 13_{10}. The table reference INFO[14] is equivalent to INFO[12.].

A table declaration should be placed on a line by itself, and may appear anywhere in the microprogram. The legal occurrences of table references in MICRO/40 assignment statements are defined in subsection 2.3.2. However, table references can only be made following table declarations. The number of table declarations is limited to 20 tables. The size of tables is limited by the number of RAM locations that are not utilized by the code of the microprogram that contains the table declarations.

2.5.8 PRELOAD

The preload pseudo operator is similar to a table declaration, except that instead of a size specification, a list of values that are preloaded into the table is specified. The table size corresponds to the number of elements in the list. The syntax of preload is:

<table preload>::= preload <table name><value list>

<table name>::= <identifier>

<value list>::= <number>|<table reference>|<value list>{,}<number>|<value list>{,}<table reference> .

The elements of the value list are considered to be assembly time constants. Hence, references to declared tables may occur in the value list.

The use of the preload pseudo operator is demonstrated by the following example:

preload DATASET 5,2,3,4, INFO[2] .

This statement causes the reservation of two control store locations to store the following five table entries:

DATASET[0]	contains the value 5,
DATASET[1]	contains the value 2,
DATASET[2]	contains the value 3,
DATASET[3]	contains the value 4,
DATASET[4]	contains the control store field address allocated for the storage of INFO[2].

The preload pseudo operator must be placed on a line by itself. The violation of this rule causes an error.

2.5.9 SET...TES

This compound pseudo operator supports microinstruction sequencing and conditional branching. As the PDP-11/40E has no microinstruction counter, each microinstruction contains the control store location of the microinstruction to be executed next in its xupf field. Conditional branches are specified in the ubf field and cause basic microbranch conditions (BUBCs) to be ORed into the low-order bits of the XUPP segment (xupf field) of the microinstruction buffer. Furthermore, the EMIT field or values from the WCS 11/40 stack may be gated onto the 11-bit EUBC BUS, in order to be ORed into the XUPP segment of the microinstruction buffer. The timing of microbranches in the PDP-11/40E is such that BUBC and EUBC address bits alter the xupf field of the microinstruction whose execution succeeds the execution of the microinstruction that contains the microbranch specification. Additionally, ORing of address bits allows only for forward conditional branches. Therefore, it is necessary that the control store locations of the expected branch destinations have identical address bits, except for those bits which may be turned on by the BUBC or EUBC address bits.

The PDP-11/40E microbranch mechanism may be enforced by using the dot pseudo operator (. = <value>) to put the microinstructions to be reached as branch destinations into the appropriate control store locations. The set... tes pseudo operator is implemented to obviate this need for explicit control store address allocation. That is, it makes the PDP-11/40E microbranch mechanism transparent for the microprogrammer. The set...tes pseudo operator is used as follows:

```
                                set
("controlled"                  {<microinstructions to be reached as
 microinstructions)            {branch destinations>
                                tes     .
```

The keywords, set and tes, must each appear on separate lines by themselves. The number of set...tes groups in a microprogram must not exceed 150.

In the assembly process, MICRO/40 counts the "controlled" microinstructions, finds an appropriate number of contiguous control store locations with 2^n address boundaries, and stores the "controlled" microinstructions into these locations. Therefore, it is illegal to assign a microinstruction that is controlled by a set...tes pseudo operator to a specific control store location (by using . = <values>). The xupf field of all microinstructions that are affected by BUBC or EUBC address bits that provide an entry into a group of "controlled" microinstructions are set to the control store address of the first microinstruction of that set...tes group. The default value of the xupf fields of all "controlled" microinstructions (which is automatically set by MICRO/40, if no field assignment is made to xupf) is the control store location of the microinstruction following the set...tes group. Note that the assembly of the set...tes pseudo operator may rearrange the microinstructions as given in the source microcode into a different ordering in the object microcode.

We may now rewrite the microprogram for adding the numbers from 1 to 10 using the set...tes pseudo operator (cf. subsection 2.3.2).

```
d_0; R[0]_d
b_10
set
   loop: d-177777+b; b_d; ubf=12
   xupf=next;<first microinstruction of whatever
             comes next>
   tes
   d_R[0]+b; R[0]_d; xupf=loop
   next: <whatever comes next>
```

Note that the field assignment, xupf=n, in the microinstruction with label, loop, can be omitted, as the appropriate microinstruction is reached by the default xupf field assignment in the set...tes group. Furthermore, the first microinstruction of <whatever comes next> may be concatenated to the microinstruction, xupf=next, in the set...tes group.

2.5.10 START...END

The basic set...tes mechanism does not lend itself to the execution of more than a single microinstruction at each branch destination. To overcome this difficulty, the start...end pseudo operator has been implemented. It can only be used within a set...tes group, i.e.,

```
set
    <"controlled" microinstructions>
    start
    <microinstructions to be executed at a single branch destination>
    end
    <"controlled" microinstructions>
tes
```

The keywords, start and end, must each appear on separate lines by themselves.

A compound set...tes - start...end construct is assembled, such that the first microinstruction of each start...end group and all single "controlled" microinstructions are placed into the block of microinstructions associated with the set...tes pseudo operator. The remaining microinstructions in the start...end groups are linked to the set...tes group by setting the xupf field of the first instructions in the set...tes group to the control store address of its start...end group successor. The xupf default value of the last microinstruction in each start...end group is the control store address of the microinstruction following the set...tes group.

```
    :
<microinstruction 0>
set
  <microinstruction 1>
  <microinstruction 2>
  start
      <microinstruction 3>
      <microinstruction 4>
      <microinstruction 5>
      set
        <microinstruction 6>
        <microinstruction 7>
      tes
      <microinstruction 8>
  end
```

(Figure 2 continued)

```
        <microinstruction 9>
    tes
    <microinstruction 10>
      ⋮
```

Figure 2: A Compound set...tes - start...end Construct

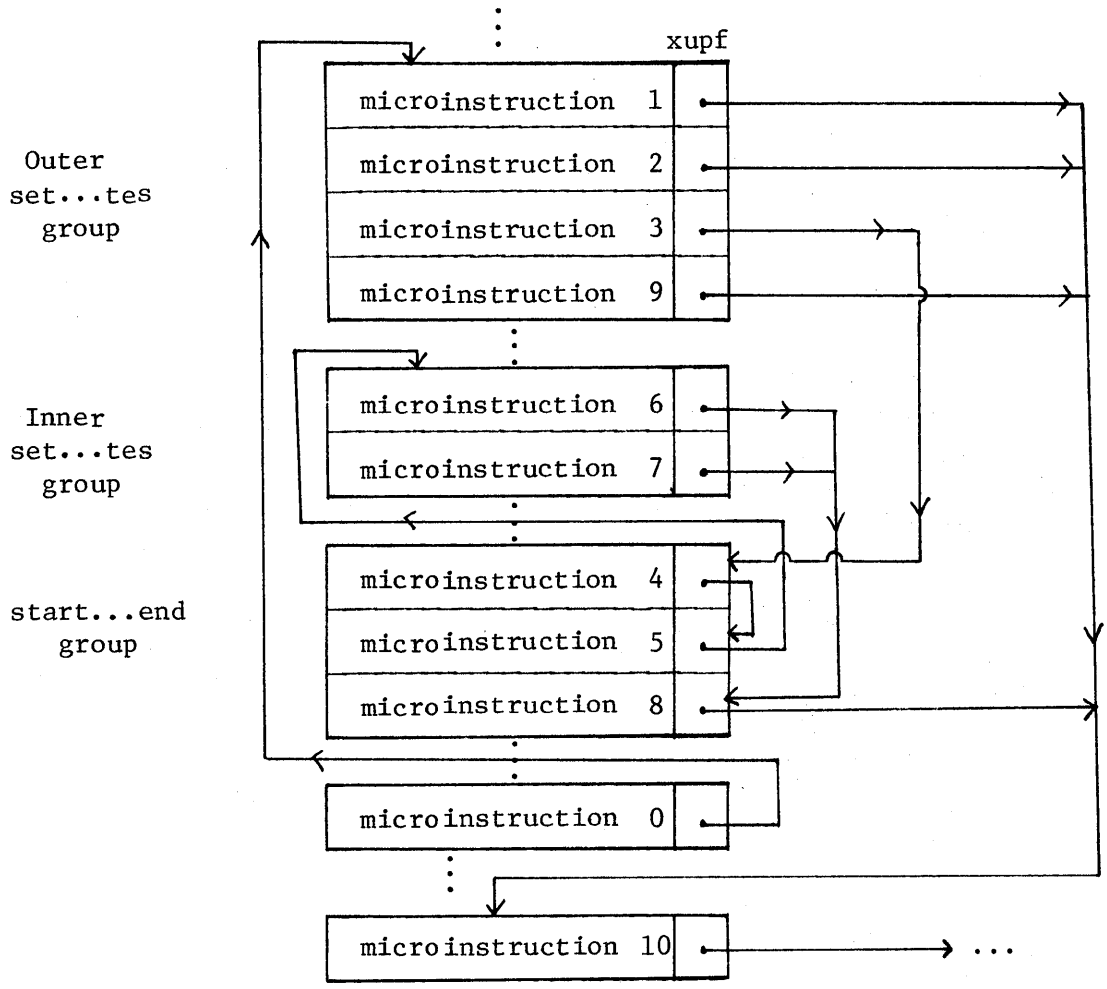


Figure 3: Control Store Allocation for a Compound set...tes - start...end Construct

The nesting of set...tes groups is generally permissible. However, they can only be nested within start...end groups, as the delay effect of conditional microbranches in the PDP-11/40E prohibits the placement of the first "controlled" microinstruction of an inner set...tes block within the outer set...tes group. That is, the pseudo operator, set, may appear anytime after the first microinstruction in a start...end group. There is no limit on the depth of set...tes group nestings, as long as the internal stack of MICRO/40 does not overflow. The control store allocation mechanism for a compound set...tes - start...end construct is illustrated in Fig.2 and Fig.3.

2.6 Macros

MICRO/40 has a macro definition facility which allows the microprogrammer to identify sequences of assembly language statements by mnemonics. The primary use of macros is to make hardware dependencies of the MICRO/40 assembly language transparent for the user by generating macro libraries in the form of common source files. Appropriate macro name selections may greatly enhance the legibility of microprograms.

2.6.1 Macro Definition

A macro definition consists of a header, a body, and a terminator. The macro header is composed of a macro name and a declaration symbol (:=). Note that macros take no parameters. The macro body is a sequence of MICRO/40 statements that does not contain a dollar sign ($) or an exclamation point(!). Any legal MICRO/40 statement is allowed in the macro body, except for macro definitions and set...tes pseudo operators. Note that macro definitions may contain other macros. The dollar sign is the macro definition terminator. The syntax of a macro definition is:

<macro definition>::= <head>:= <body>$

<head>::= <identifier>

<body>::= <any sequence of MICRO/40 statements with the exceptions given above> .

Fig.4 shows a listing of the MICRO/40 source file, defs. mic, which includes macro definitions which have been found to be of general use.

```
! standard definitions for micro -- 11 October 1974
!                                        rev: 19 November 1974
!                                        rev:  7 December 1974
!                                        rev: 11 June 1975
!
r0 := r[0]$;    r1 := r[1]$;    r2 := r[2]$;    r3 := r[3]$
r4 := r[4]$;    r5 := r[5]$;    r6 := r[6]$;    r7 := r[7]$
r10 := r[10]$;  r11 := r[11]$;  r12 := r[12]$;  r13 := r[13]$
r14 := r[14]$;  r15 := r[15]$;  r16 := r[16]$;  r17 := r[17]$
rsp := r[6]$;   rpc := r[7]$;   rdf := r[df]$;  rsf := r[sf]$
temp := r[10]$; rsrc := r[11]$; rdst := r[12]$
rir  := r[13]$; vect := r[14]$; temc := r[15]$
spus := r[16]$; adrsc := r[17]$;        rba := r[ba]$
dati := bus=1$;         dato := bus=5$
datip := bus=3$;        datob := bus=7$
p1 :=clk =2$;   p2 :=clk =4$;   p3 :=clk =6$
exit := xupf = 16$  ! return to rom
begin := beg: .=2000;$
goto := xupf =$;        case := eubc_$; popst := d_s$
but := ubf =$; skipzero := ubf = 12$ ! skip on d = 0
return := eubc_s$;      endproc := xupf=0$
smod := 11:9$;  dmod := 5:3$;   prop := cp=1$
! end of macros
! ADDITIONAL MACROS
POP:=DEST=1;MSC=4$
PUSH:=DEST=1;MSC=3$
```

Figure 4: defs.mic

2.6.2 Macro References

Macros are referenced by their names. They may be referenced anywhere in a microprogram, as long as they are preceded by their definition in the MICRO/40 source file. Upon reference, the macro name is substituted by its macro body. This substitution is a strict text substitution. After a macro is expanded, the macro text is checked again for macro references. If further macro references are encountered, they are substituted, before the assembler proceeds processing of the actual source microcode.

All macros that are referenced within a macro must be defined outside that macro, as macro definitions are not allowed within macros. Furthermore, recursive macro expansion is not permissible. Macro expansions do not place a delimiter at the end of the expanded text. Hence, it is possible to concatenate MICRO/40 statements across the macro expansion. That is, a single microinstruction may be generated from a combination of macros and MICRO/40 statements, or macros may be referenced within MICRO/40 statements. Obviously, the control store allocation mechanism of the set...tes pseudo operator (cf. subsection 2.5.9) is not supported by this macro expansion technique, and thus, the use of set...tes groups in macros is not permissible.

2.6.3 Common Source Files

It is often desirable to include the same source microcode into several assemblies. This is especially true for macro files of a macro library. For example, the file, defs.mic, should be included into every source file to make microprograms more legible. To this end, the 'require' statments has been implemented.

The syntax for the use of common source files in a microprogram is:

<common source file>::= require <file name> ,

where <file name> is the name of a MICRO/40 source file. The require statement may occur at any point in a source microprogram, if it is placed on a line by itself. However, it is not permissible to use require statements within set...tes or start...end groups. When a require statement is encountered, the microassembler replaces it by the specified MICRO/40 source file. After the substitution of the require statement, the input to the microassembler from the original file is resumed. The 'require' file may contain further require statements. That is, the strict text substitution for require statements allows their nesting to an arbitrary level. However, this substitution mechanism prohibits set...tes or start...end groups to be open across

'require' files.

To illustrate the use of macros and common source files we revisit the microprogram for adding the numbers from 1 to 10 (cf. subsection 2.5.9).

```
require defs.mic
d_0; r0_d
b_10
set
  loop: d_177777 +b; b_d; skipzero
  goto next; <first microinstruction of whatever comes next>
tes
d_r0+b; r0_d; goto loop
next: <whatever comes next>
```

3. Features of MICRO/40

In this section, some of the more subtle features of the PDP-11/40E are outlined, that should be remembered when microcoding. Some microprogramming techniques that proved to be of general use [1] are described. However, as mentioned in the introduction, microprogramming requires careful algorithm design and coding on the basis of a solid understanding of both the algorithm and the machine. This is especially true for machines with horizontal microinstructions, where any arbitrary bit pattern may be assigned to a microinstruction, and the machine tries to execute any of these bit patterns. The microprogram testing and debugging facilities in our microprogramming support system are not comprehensive, and may not be sufficient to scope with unpredictable and hardware-dependent microprogram errors.

3.1 Microinstruction Timing

One of the major objectives in the development of the WCS 11/40 was to retain the processor cycle times of the basic PDP-11/40 processor. Therefore, the three processor clock cycles length, CLKL 1, CLKL 2, and CLKL 3, as provided by the PDP-11/40 timing control logic have been adopted for the execution of microinstructions from the WCS 11/40 control stores. The processor clock cycles are defined as follows:

CLKL 1 generates a P1 pulse 140 ns after the start of the microinstruction execution,

CLKL 2 generates a P2 pulse 200 ns after the start of the microinstruction execution,

CLKL 3 generates a P2 pulse and a P3 pulse 200 ns and 300 ns after the start of the microinstruction execution, respectively.

A detailed description of PDP-11/40E timing characteristics is given in [9]. Table 1 associates permissible processor clock cycle times with the basic carrier-to-carrier transfer in the PDP-11/40E.

Carrier-to-Carrier Transfer	Permissible Clock Cycles
B ← DMUX BUS	P1, P3
B ← SMUX	P3
PS ← DMUX BUS	P1, P3
PS ← SMUX	P3
R[i] ← DMUX BUS	P1, P3
R[i] SMUX	P3
IR ← DMUX BUS	P1, P3
IR ← SMUX	P3
CLK PS(C)	P1, P3
CLK PS(N,Z,V)	P1, P3
CLK PS(N,Z,V,C)	P1, P3
BA ← BA MUX	P1, P2
D ← ALU	P2
D(C) ← COUT MUX	P2
D ← ALU; DMUX BUS ← D	P3
ALU CIN ← CPFF	P3
CPFF ← COUT MUX[SCOM]	P3
TOS ← EMUX	P1, P2, P3
TOS ← RD BUS	P2, P3
SP ← SP+1	P1, P2, P3
SP ← SP-1	P1, P2, P3
SP ← XU<75:72>	P1, P2, P3
SP ← SP-1; S ← EMUX	P2, P3
SP ← SP+1; SMUX ← S	P1, P2, P3
EUBC ← SMUX	P2, P3
RAM ← DMUX	P3
SMUX ← RAM	P3

Table 1: Carrier-to-Carrier Transfer Timing

The microassembler analyzes the microoperations (carrier-to-carrier transfers) specified in an input line and assigns a proper processor clock cycle for the execution of that microinstruction. If the processor cycle time is set by the microprogrammer, the microassembler check is skipped. Thus, an incorrect processor cycle time setting is not detected by MICRO/40, and no error message is given. This feature of the microassembler is due to the fact that microinstructions are assembled individually, but the appropriate processor cycle times may depend on preceeding microinstructions. For example, the microinstruction following a microinstruction that contains the clkoff pseudo operator must have a CLKL 1 or CLKL 3 processor clock cycle. Although it is advisable (and convenient) to let the microassembler assign appropriate processor clock cycle lengths, the microprogrammer should know the execution time requirements for each microinstruction to be able to determine the relative performance of alternative microinstruction sequences.

3.2 UNIBUS Control

The PDP-11/40 processor control is a combination of synchronous and asynchronous operations. Carrier-to-carrier transfers as discussed in the preceeding subsection are synchronous operations. Asynchronous timing conditions evolve in UNIBUS operations and are controlled by the UNIBUS timing and control logic. To synchronize UNIBUS operations with synchronous precessor operations, the processor clock may be turned off upon microinstruction execution and restarted by the UNIBUS timing and control logic (for more detail, cf. [1], [8], [9]).

The microprogrammer must explicity control UNIBUS operations, which allow uniform access to main memory and peripheral registers. On the one hand, explicit UNIBUS control provides an opportunity to optimize the implementation of machine language instructions which involve UNIBUS transfers. On the other hand, it represents a source of timing errors.

UNIBUS operations are controlled by the microoperation field, BUS=CBUS, BGBUS, with CBUS= C1,C0 (cf. subsection 2.2.1). The BUS field definitions are given below.

BUS	C1	C0	BG	Operation
0	0	0	0	not defined
1	0	0	1	DATI (word operation)
2	0	1	0	await BUS BUSY
3	0	1	1	DATIP (read-modify-write)
4	1	0	0	not defined
5	1	0	1	DATO (word operation)
6	1	1	0	restart on peripheral release
7	1	1	1	DATOB (byte operation)

C1 and C0 determine which of the four possible read/write operations will occur and BGBUS initiates the action. During UNIBUS operations, the UNIBUS address is held in the BA Register. Output data are stored in the D Register, and input data are received at D MUX.

3.2.1 UNIBUS READ Operations

For a READ operation (data input), the UNIBUS address must be available in the BA Register, when the DATI UNIBUS control code is asserted. Succeeding microinstructions may be executed while the UNIBUS READ is carried out, as long as they do not modify the BA Register or assert another UNIBUS control code. However, it is necessary to set clkoff, before the input data are accepted. Upon completion of the READ operation, the data will be present on the UNIBUS and the processor clock is restarted. If the UNIBUS completes its read cycle before a clkoff is asserted, the processor clock does not stop. The microinstruction following the microinstruction containing the pseudo operator clkoff must have a CLKL 1 or CLKL 3 processor clock cycle, and must pull the input data off the UNIBUS immediately.

A typical UNIBUS READ cycle has the following form (cf. Fig. 4).

```
    ba_R[i]; dati !BUS=1, put UNIBUS address into BA
    {any number of microinstructions that do not modify BA or
     assign a value to the BUS field}
    <last microinstruction>; clkoff
R[j]_ unibus; <other statements> !CLKL 1 or CLKL 3 clock cycle
```

As a more specific example, we may consider the following microinstruction sequence for popping the PDP-11/40 main memory stack.

```
ba_R[6]; dati  ! read top of stack
d_R[6] + 2; R[6]_d; clkoff ! increment stack pointer
R[2]_unibus
```

The DATI UNIBUS control code is used for both word and byte operations. It always returns a 16-bit word to D MUX. To select a byte the main memory address must be tested. If this address is even (BA<0>= 0), the byte is in the lower half of the word. If this address is odd (BA<0>=1), the byte is in the upper half of the word. BUT 35 (ubf=35) may be used to perform the address test. Alternatively, the assignment statement, EUBC ← TOS<0>, may be used, if the UNIBUS address is stored in TOS. The B Register, together with B MUX, may be used to align the appropriate byte.

3.2.2 UNIBUS WRITE Operations

UNIBUS WRITE operations are similar to READ operations. The following differences should be observed. For a WRITE operation (data output), the UNIBUS address must be available in the BA Register and the output datum must be clocked to the D Register, when the DATO UNIBUS control code is asserted. Succeedingly executed microinstructions must keep the BA Register and the D Register constant.

A typical UNIBUS write cycle has the following form (cf. Fig. 4).

```
ba_R[i]  ! put UNIBUS address into BA
d_R[j]; dato  ! BUS=5, put datum into D
{any number of microinstructions that do not modify BA
 and D, or assign a value to the BUS field}
<last microinstruction>; clkoff
<any microinstruction with a CLKL 1 or CLKL 3 clock cycle>
```

As a specific example, we consider the following microinstruction sequence for pushing the PDP-11/40 main memory stack.

```
d,ba_R[6]-2; R[6]_d ! decrement stack pointer
d_R[2]; dato; clkoff  ! write to top of stack
```

The DATOB UNIBUS control code specifies a byte WRITE operation. The microinstruction for the implementation of a byte WRITE operation is similar to that for DATO. It is the microprogrammer's responsibility to place the byte to be written into the proper byte of the D Register. As a safeguard for the case that the main memory address may either be even (low-order byte) or odd (high-order byte), the byte to be written should be duplicated in both bytes of the D Register. The B Register, together with B MUX, may be used for this purpose.

3.2.3 UNIBUS READ - MODIFY - WRITE Cycle

The DATIP UNIBUS control code specifies a READ - MODIFY - WRITE cycle. It uses the address in the BA Register both for the READ and the WRITE operation. A microinstruction sequence for a UNIBUS read-modify-write cycle is given below. This microinstruction sequence implements a modification of the top of the PDP-11/40 main memory stack.

```
ba_R[6]; datip; clkoff  ! put UNIBUS address into BA
                        ! BUS=3
R[10]_unibus            ! read top of stack
d_R[10]+1; dato; clkoff ! modify top of stack, BUS=5
<any microinstruction with a CLKL 1 or CLKL 3 cycle>
```

3.2.4 Exceptional Conditions

The ability to perform UNIBUS operations and execute microinstructions in parallel provides the microprogrammer with the opportunity to keep the processor totally UNIBUS-bound, while doing processing in addition to UNIBUS control. Care must be taken that all initiations of UNIBUS operations are followed by a clkoff. Otherwise, the UNIBUS becomes locked. On the other hand, setting clkoff, without setting BGBUS halts the processor and requires a restart from the processor console, except for interrupt sequences.

Whenever an odd address is clocked into the BA Register, or the DATOB UNIBUS control code is asserted, the following two conditions must be met. First, the instruction register must contain a valid machine language byte instruction. Second, bit 0 of the DAD field must be set (allow odd address and DATO for byte instruction). If these conditions are not met, a jam into the 8 low-order bits of XUPP (segment of the microinstruction buffer) occurs and the flow of control in the user microcode will be distorted. This effect is due to the fact that the jam addresses are associated with the standard PDP-11/40 emulator microinstructions, not with the user microporgram.

3.3 Data Flow

The following idiosyncrasies which are caused by the PDP-11/40E data paths and functional hardware units should be remembered when writing micro-code.

3.3.1 RD BUS

As discussed in section 2.3.3, the result of gating several independent sources onto the RD BUS is the ORing of the source contents. This feature of

the RD BUS may be exploited to produce a positive effect, when a table lookup is to be performed into a table of fixed main memory locations that is indexed by a general purpose register.

For example, let R[0] be assumed to hold the table index, which must be even for word addressing. Let BASE be the base address of the table in main memory. Then, the following microinstruction implements a table access,

```
ba_R[0]; dest=0; msc=1; emit=BASE; dati      .
```

The field assignments, dest=0, msc=1, specify the transfer RD BUS ← EMIT. Hence, the content of R[0] and the EMIT field are ORed on the RD BUS, i.e., RD BUS ↔ R[0] or BASE. BASE should be chosen such that it has at least as many zeros in its low-order bits as there are non-zero low-order bits in the binary representation of the largest table index, i.e., such that R[0] + BASE = R[0] or BASE. Then, the value clocked into the BA Register is the effective word address of a table entry, R[0] + BASE = <index> + <base address>.

To obtain the same result, without using the ORing feature of the RD BUS, requires the following two microinstruction

```
d_R[0]; b_d
ba_BASE + B; DATI      .
```

3.3.2 CONSTANTS

The WCS 11/40 provides the microprogrammer with arbitrary arithmetic and addressing constants via the EMIT field. However, EMIT field constants must be A-inputs to the ALU or must temporarily be stored in the B Register to be usable as ALU B-inputs. The latter mechanism requires the execution of two consecutive microinstructions for the use of an EMIT field constant at the ALU. This difficulty can often be overcome by using one of the B CONSTANTS as provided by the basic PDP-11/40E processor. The use of B CONSTANTS instead of EMIT field constants at the ALU B-input may save up to 300 ns in the execution of the associated microinstruction sequence.

3.3.3 Instruction Register

The PDP-11/40 processor is implemented for the specific PDP-11/40 machine instruction set and, as a consequence, is not of general-purpose nature. The basic machine instruction decoding generates basic microbranch codes (BUBCs) for several branch microtests (BUTs), signals required by the microbranch control logic, the condition code control logic, and the ALU control logic. Hence, the contents of the instruction register (IR) may affect the data paths and the ALU, as they are not exclusively controlled by the micro-

instruction. Some PDP-11/40 machine language instructions (e.g., SBC, RESET, MFPI,MTPI) may even have an effect on the data paths, when the instruction register is clocked the next time [8]. Therefore , great care must be taken, if IR is used by user microprograms. In user microprograms, it is advisable to decode machine instruction from TOS by using the WCS 11/40 shift/mask unit to pull various instruction fields from TOS onto the EUBC BUS(for conditional branching).

3.4 Control Flow

In this section, we discuss basic mechanisms for the sequencing of micro-instruction executions.

3.4.1 Microinstruciton Execution

With the last pulse edge of each processor cycle, a microinstruction is gated into the microinstruction buffer. Thus, the interval between the loading of two seccessive microinstructions into the microinstruction buffer is a function of the processor cycle length of the first microinstruction. This organization requires that the execution (E) of the current microinstruction and the fetch (F) of the successor microinstruction are overlapped in time. This overlap is illustrated in Fig. 5.

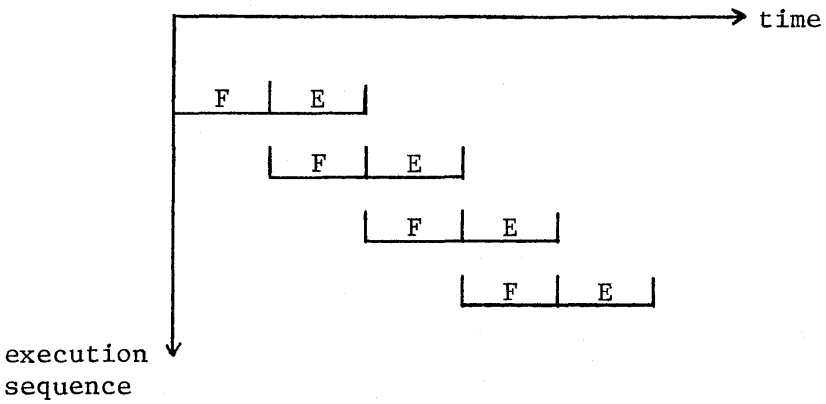


Figure 5: Microinstruction Fetch/Execute Overlap

3.4.2 Unconditioned Microinstruction Sequences

MICRO/40 handles unconditioned microinstruction sequences by assigning control store addresses to the xupf fields in microinstructions. The basic rule is to assign consecutive addresses to successive microinstructions, unless a 'goto' (xupf field assignment) is assembled. In the latter case, the control store address specified in the 'goto' is assigned to the xupf field. Hence, at the beginning of the execution of a microinstruction with an unconditioned

successor., the address of the next microinstruction is available in the XUPP segment of the microinstruction buffer.

3.4.3 Conditioned Microinstruction Sequences

Whenever a microinstruction is clocked into the microinstruction buffer, XUPP is modified by ORing the 11 bits of the EUBC BUS and the six BUBC bits (into the six low-order bits of XUPP). The modified XUPP is immediately used to address the successor microinstruction. At this point, an alteration of the address of the successor microinstruction as specified in the current microinstruction has not occurred. Hence, it is impossible for a micro-instruction to influence the address of its successor. Instead, conditional branching is performed by setting the EUBC/BUBC lines such that the address in the xupf field of the successor microinstruction are altered. This organ-ization of conditional microbranches is illustrated in Fig. 6.

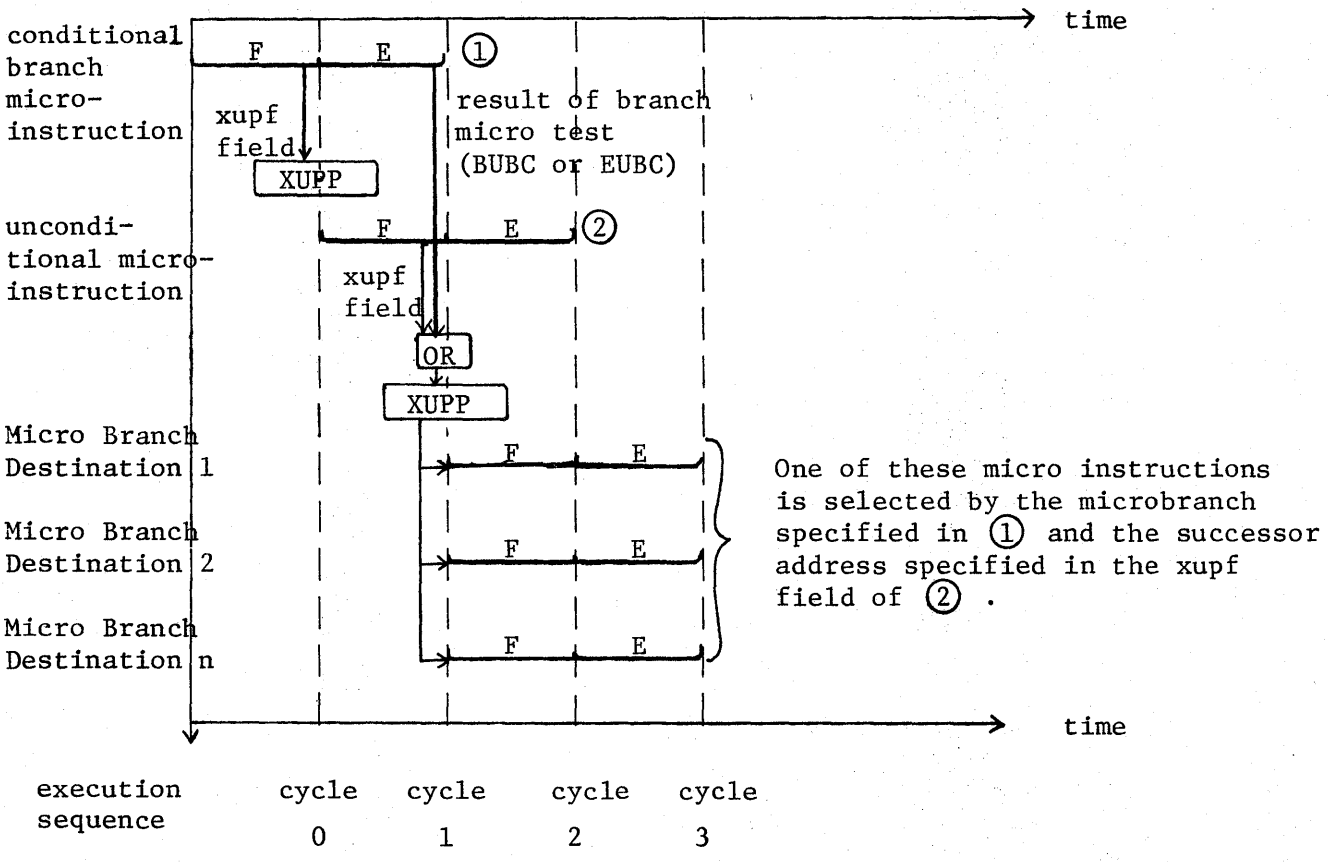


Figure 6: Conditional Microbranching

The delayed microbranching in the PDP-11/40E is unnatural to most programmers and requires great care, as it makes microprograms difficult to modify, because of the interdependence of microinstructions in a branching sequence. Furthermore, since conditional branching is due to ORing into XUPP, the proper bits in XUPP must be 0 for the ORing to have the right effect. MICRO/40 supports the microprogrammer in satisfying this requirement by the provision of the set...tes pseudo operator. A set...tes construct is assembled by allocating a block of RAM locations with 2^n address boundaries for the storage of the microinstructions at the branch destinations. As MICRO/40 allocates control store blocks for all set...tes groups, before it assigns RAM locations to the rest of the microinstructions, the base addresses of set...tes blocks can be chosen to include an appropriate number of 0's.

The PDP-11/40E provides the following two mechanisms for setting the bits to be ORed into XUPP. Examples for the application of these mechanisms are discussed in the next two subsections.

BUT

The branch micro test (BUT) is a feature of the basic PDP-11/40. It consists of about thirty different tests which are invoked by field assignments to ubf (cf.[1], [8]). Branch micro tests detect particular processor states and, in response, set appropriate values on the BUBC lines. The generally useful BUTs are:

ubf	BUT	BUBC
12	D=0	000001
16	interrupt	000001

EUBC

The extended microbranch control (EUBC) is based on the shift/mask unit in the WCS 11/40. Using this field extraction unit, arbitrary contiguous fields of the word at the top of the stack, the EMIT field, or any 16-bit RAM field can be gated onto the 11-bit EUBC BUS.

3.4.4 IF Statement

A simple IF statment,

IF<condition>THEN<microinstructions1>ELSE<microinstructions2>,

may be implemented using the BUT mechanism or the EUBC mechanism.

The following timing characteristics must be taken into account, when a BUT 12 is used. If the microinstruction containing BUT 12 has a CLKL 1 or

CLKL 2 processor clock cycle, then the value of the D Register at the beginning of the microinstruction execution is tested. If the microinstruction has a CLKL 3 clock cycle, the value clocked into the D Register with the P2 pulse is tested. The effect of this timing condition is demonstrated by the following two implementations of an IF statement with BUT 12.

(1)
```
d_R[0]
d_R[1]; but 12
b_d
set
   {start}       !R[0]≠0
         <microinstructions 1>
   {end}
   {start}       !R[0]=0
         <microinstruction 2>
   {end}
tes
   :
```

The use of the start...end pseudo operator is optional, as indicated by the meta symbol, { }. The microinstruction, d_R[1]; but 12, is executed in a CLKL2 processor clock cycle (cf. Table 1), and hence, but 12 tests the content of the D Register at the beginning of microinstruction execution. At this point, D contains the value of R[0]. Thus, (1) implements the IF statement,

IF R[0]=0 THEN<microinstructions 2>ELSE<microinstructions 1>.

If D=0, BUT 12 causes a 1 to be ORed into xupf<0> of the microinstruction b_d, control is transferred to the second element in the set...tes group. Otherwise, BUT 12 generates BUBC=0 and control is transferred to the first element in the set...tes group (cf. Fig.3).

(2)
```
d_R[0]
d_R[1]; b_d; but 12
hoop
set
   {start}         ! R[1]≠0
         <microinstructions 1>
   {end}
   {start}         ! R[1]=0
         <microinstructions 2>
   {end}
tes
```

The noop is required to account for the delayed conditional branching in the PDP-11/40E. The microinstruction, d_R[1]; b_d; but 12, is executed in a CLKL 3 processor clock cycle (cf. Table 1). As a consequence, the content of D is tested at P2 time. At this point, R[1] is clocked into D. Thus, (2) implements the IF statement,

IF R[1]=0 THEN <microinstructions 2>ELSE <microinstructions 1>.

The following implementation of the IF statement uses the EUBC mechanism.

(3)
```
tos_R[0]
eubc_tos<15>
noop
set
   {start}     ! R[0]≥0
         <microinstructions 1>
   {end}
   {start}     ! R[0]<0
         <microinstructions 2>
   {end}
tes
   .
   .
   .
```

In this implementation R[0] is written onto the WCS 11/40 stack. From there, the sign bit (TOS<15>) is extracted using the shift/mask unit and gated onto the EUBC BUS<0>. Hence, if R[0]≥0, EUBC BUS<0>=0, and the first element in the set...tes group is reached, as the xupf field of the microinstruction, noop, is not modified. Otherwise, EUBC BUS<0>=1 and control is transferred to the second element in the set...tes group. Thus, (3) implements the IF statement,

IF R[0]≥0 THEN<microinstructions 1>ELSE<microinstructions 2>.

3.4.5 CASE Statement

The CASE statement extends the IF statement as to multiple branch destinations. The importance of CASE statements in microprogramming stems from the fact that multi-way branches are vital to machine language instruction decoding. The general form of a CASE statement is:

```
CASE<expression>DO
<microinstructions 1>
<microinstructions 2>
        .
        .
        .
<microinstructions n>
```

The expression in the CASE statement specifies an index, i, that selects <microinstructions i> to be executed. For the implementation of CASE statements, the BUT mechanism, the EUBC mechanism,or a combination of both may be used.

The following example illustrates the use of the EUBC mechanism for the implementation of a CASE statement. Assume an 8-way branch is to be implemented to decode the op-code bits R[13]<15:13> of machine instructions stored in R[13]. To this end, the op-code bits R[13]<15:13> can directly be used as the expression in an appropriate CASE statement.

```
tos_R[13]
eubc_TOS<15:13>
noop
set
   start     ! op-code 000
        <microinstructions 0>
   end
   start     ! op-code 001
        <microinstructions 1>
   end
    :
   start     ! op-code 111
        <microinstruction 7>
   end
tes
```

The following microinstruction sequence implements a 4-way branch using a combination of the BUT mechanism and the EUBC mechanism. Here, the cases R[0]>0, R[0]=0, and R[0]<0 are distinguished, whereas the forth case, R[0]=0 and R[0]<0, is impossible in the 2's-complement number representation of the PDP-11/40E.

```
d_R[0]; tos_d        ! CLKL 3 clock cycle
eubc_tos<15>^1; but 12
noop
set
   start  ! R[0]>0, EUBC BUS<1>= 0, BUBC=0
        <microinstructions 1>
   end
   start  ! R[0]=0, EUBC BUS<1>=0, BUBC=1
        <microinstructions 2>
```

```
            end
            start  ! R[0]<0, EUBC BUS<1>=1, BUBC=0
                 <microinstructions 3>
            end
            start  ! impossible, EUBC BUS<1>=1, BUBC=1
                   noop
            end
        tes
```

Note that the sign bit, TOS<15>, is gated onto EUBC BUS<1>, such that EUBC BUS<1> and BUBC<0>are independently ORed into XUPP<1:0>.

3.4.6 Micro Subroutines

Analogous to software programming, subroutines are crucial to microprogramming. Normal, nested, and recursive micro subroutines are easily implemented using the WCS 11/40 stack and the EUBC branch mechanism. A subroutine call pushes the return address, retadd, on the WCS 11/40 stack and sets the xupf field of the calling microinstruction to the subroutine address, subr. The return from subroutine is implemented by poping the WCS 11/40 stack in the second-to-last microinstruction in the subroutine and gating the return address, retadd, onto the EUBC BUS. The xupf field of the microinstruction executed last in the subroutine must be set to 0 so that the return address can be ORed into XUPP, to form the effective address of the next microinstruction.

```
  call: s←retadd; goto subr  ! subroutine call
retadd: <whatever follows next>
              :
              :
  subr: <first microinstruction of the subroutine>
                  :
                  :
        <second-to-last microinstruction>; eubc_s  } return from
        <last microinstruction>; xupf=0            } subroutine
```

For the implementation of a recursive subroutine, it is important that the recursive subroutine call occurs in a set...tes group (conditional branching) in order to prevent an infinite recursion. Furthermore, it is the microprogrammer's responsibility to prevent stack overflow or underflow, as stack control is not supported by the WCS 11/40 hardware. An example of a schema for the implementation of recursive subroutines is given below.

```
  call: s_retadd 1; goto recurs  ! subroutine call
retadd: <whatever follows next>
              :
              :
```

```
reurs: <first microinstruction in the recursive subroutine>
              :
              :
       <conditional branch microinstruction>
       set
           s_retadd 2; goto recurs  ! recursive subroutine call
           <microinstruction reached at the end of the recursion>
       tes
       retadd 2: <whatever comes next>
              :
              :
       <second-to-last microinstruction>; eubc_s  }  return from
       <last microinstruction>; xupf=0            }  subroutine
```

The data objects of microprograms are the contents of registers or main memory locations. All these carriers are by nature global to the hardware of the PDP-11/40E and can be accessed by any microinstruction. Therefore, the PDP-11/40E does not provide a micro subroutine parameter passing mechanism. Various parameter passing mechanisms may be implemented using, for example, the general purpose registers, the WCS 11/40 stack, the table or preload pseudo operators, the EMIT field, or the main memory table look-up described in subsection 3.3.1.

3.5.7 Exit from WCS 11/40 Control Store

The interface between the basic PDP-11/40 processor and the WCS 11/40 is organized such that the extension is turned off, whenever the XUPP (xupf field) segment of the microinstruction buffer is assigned a value less than 400_8. At this point, control is automatically transferred to the standard PDP-11/40 emulator ROM. Consequently, the last microinstruction executed in the RAM, which exits to the ROM, must not use the extension hardware.

The execution of microcode from the WCS 11/40 control stores is usually independent of external PDP-11/40 processor conditions. Therefore, special provision should be taken, when control is returned to the standard PDP-11/40 emulator. It is recommended to check for conditions that could have caused a PDP-11/40 interrupt, while the PDP-11/40E was controlled from the WCS 11/40. BUT 16 detects any condition that would cause an interrupt at the next PDP-11/40 machine instruction fetch. Thus, the following micro-instruction sequence is appropriate for transferring control from the WCS 11/40 control stores to the standard PDP-11/40 emulator ROM:

<second-to-last microinstruction>; but 16
<last microinstruction>; goto 16 .

If no interrupt occurred, control is transferred to ROM location 16 for the next PDP-11/40 machine instruction fetch. Otherwise, control is transferred to ROM location 17, where an interrupt service routine is initiated.

4. Operating MICRO/40

MICRO/40 runs under the UNIX operating system and is invoked by a UNIX command. The UNIX editor [2] is used to generate MICRO/40 source files. MICRO/40 source files must be stored in UNIX text files so that they can be used as arguments in the UNIX command that invokes the MICRO/40 assembler.

4.1 Invoking MICRO/40

MICRO/40 is invoked by the UNIX command,

mic{<opt>}<name>.mic .

The name of a MICRO/40 source file must be of the form,

<name>.mic

where <name> is any legal UNIX text file name. The suffix, mic, indicates that the source is written in MICRO/40.

In this section, we refer to the example microprogram[2], called fastc.mic. This microporgram implements two PDP-11/40 machine language subroutines that handle the environment switch for subroutine calls in the "C" programming language of UNIX [4]. It saves and restores registers that are used for parameter passing in subroutine calls/returns. Further details of fastc.mic will be introduced as needed. A listing of the source file of the microprogram, fastc.mic, is shown in Fig. 7.

---

2) This microprogram was developed by K. Bullis, J. Bjoin, and T. Lunzer as a course project for (H. K. Berg) CSci 5299, Microprogramming, Winter Quarter 1978.

```
require defs.mic

begin noop
.=2001; d_210; b_d
d_rir-b !compare instruction
skipzero
d_211; b_d

set

  start          !check for other instr
    d_rir-b
    skipzero
    noop

    set

        goto 150

      start      !211 instr
        d_r5     !r1<-r5
        r1_d
        d,ba_r1-2; r1_d !pop r4
        dati; clkoff
        r4_unibus
        ba,d_r1-2; r1_d !pop r3
        dati; clkoff
        r3_unibus
        ba,d_r1-2; r1_d !pop r2
        dati; clkoff
        r2_unibus
        ba,d_r5 !sp<-r5
        r6_d; dati; clkoff
        r5_unibus          !r5<-(sp)+
        d_r6+2; r6_d
        ba_r6; dati        !rts pc
        d_r6+2; r6_d; clkoff
        r7_unibus; but 16
        goto 16
       end

    tes
  end

start          !210 instr
      d,ba_r6-2; r6_d !push r5
      d_r5; data; clkoff
      d_r7; p3       !r5<-r7
      r5_d
      r0_d     !r0<-r5
      d_r6     !r5<-r6
      r5_d
      d,ba_r6-2; r6_d !push r4
      d_r4; data; clkoff
      d,ba_r6-2; r6_d !push r3
      d_r3; data; clkoff
      d,ba_r6-2; r6_d !push r2
      d_r2; data; clkoff
      d_r6-2; r6_d     !r6<-r6-2
      d_r0     !r7<-r0
      r7_d; but 16
      goto 16
 end

tes

finis
```

Figure 7: fastc.mic

In the command, mic{<opt>}<name>.mic, <opt> denotes one of the three optional assembly flags, -a, -s, and -d. These optional assembly flags are directives to the microassembler which affect the assembly process in the following way

-a The microprogram source is assembled and a pseudo-readable form of the object microcode is stored in a file called <name>.ass. This file is in UNIX assembler format. From this file, the binary version of the assembled microcode can be generated using the UNIX assembler as a post-processor of MICRO/40.

-d The microprogram source is assembled and a pseudo-readable form of the object microcode is stored in a file called <name>.dec. This file is in DEC assembler format, so that the binary version of the assembled microcode can be generated using the PDP-11/40 DEC assembler as a post-processor of MICRO/40.

-s The microprogram source is assembled and the micro-simulator [5] is called, if no assembly errors occured.

If the microassembler detects an assembly flag other than -a, -d, or -s, a warning is issued and the assembly flag is ignored. If both flags, -a and -d, are given in the microassembler invokation, only the last (right-most) flag is accepted. If the assembly flag, -s, is specified, all other flags are ignored.

4.2 MICRO/40 Output

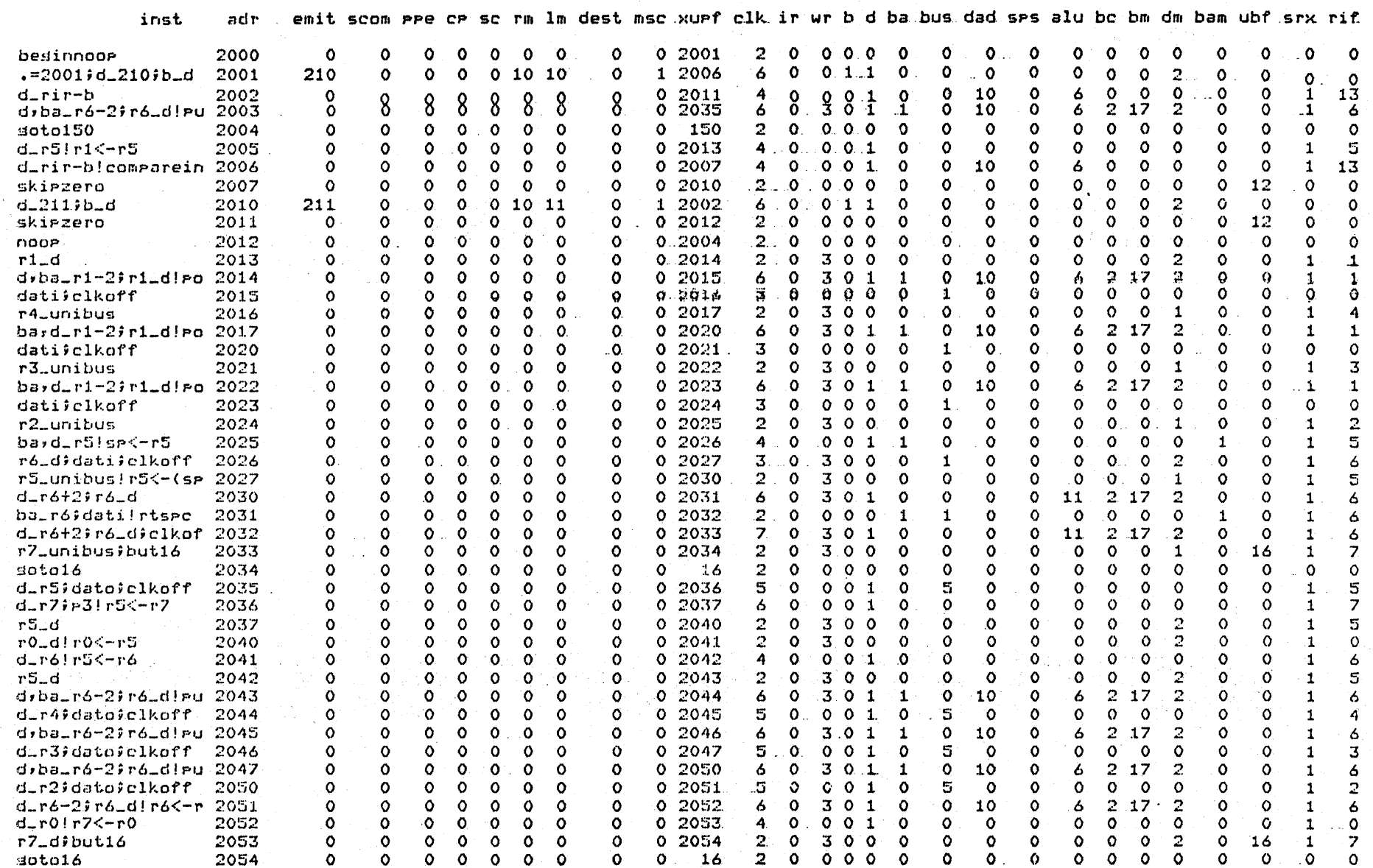
When an assembly is terminated and no assembly errors occurred, MICRO/40 generates three files, namely, <name>.lst, <name>.bin, and <name>.tab.

<name>.lst

This file is a listing of the microprogram object code in the 80-bit PDP-11/40E microinstruction format, followed by a list of mnemonic labels and their associated control store addresses. The listing contains the first 18 characters of each source code line and the assigned control store addresses of the corresponding microinstruction. In the representation of the object code, the content of the individual microoperations fields is given by octal numbers (cf. subsection 2.2). Due to space limitations, abbreviations are used for some of the microoperation field names:

rm	↔ rml,	ba	↔ clkba,
lm	↔ lml,	bc	↔ sbc ,
ir	↔ clkir,	bm	↔ sbm ,
b	↔ clkb,	dm	↔ sdm ,
d	↔ clkd,	bam	↔ sbam .

The file, fastc.lst, is listed in Fig. 8. Note that the order of the micro-instructions in fastc.lst deviates from the order of the microinstructions in fastc.mic (cf.Fig.7). This reordering is due to the fact that MICRO/40 allocates control store locations for the set...tes groups (cf. subsection 2.5.10)



inst	adr	emit	scom	ppe	cp	sc	rm	lm	dest	msc	xupf	clk	ir	wr	b	d	ba	bus	dad	sps	alu	bc	bm	dm	bam	ubf	srx	rif
beginnoop	2000	0	0	0	0	0	0	0	0	0	2001	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
.=2001;d_210;b_d	2001	210	0	0	0	0	10	10	0	1	2006	6	0	0	1	1	0	0	0	0	0	0	0	2	0	0	0	0
d_rir-b	2002	0	0	0	0	0	0	0	0	0	2011	4	0	0	0	1	0	0	10	0	6	0	0	0	0	0	1	13
d;ba_r6-2;r6_d!pu	2003	0	0	0	0	0	0	0	0	0	2035	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	6
goto150	2004	0	0	0	0	0	0	0	0	0	150	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d_r5!r1<-r5	2005	0	0	0	0	0	0	0	0	0	2013	4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	5
d_rir-b!comparein	2006	0	0	0	0	0	0	0	0	0	2007	4	0	0	0	1	0	0	10	0	6	0	0	0	0	0	1	13
skipzero	2007	0	0	0	0	0	0	0	0	0	2010	2	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0	0
d_211;b_d	2010	211	0	0	0	0	10	11	0	1	2002	6	0	0	1	1	0	0	0	0	0	0	0	2	0	0	0	0
skipzero	2011	0	0	0	0	0	0	0	0	0	2012	2	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0	0
noop	2012	0	0	0	0	0	0	0	0	0	2004	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r1_d	2013	0	0	0	0	0	0	0	0	0	2014	2	0	3	0	0	0	0	0	0	0	0	0	2	0	0	1	1
d;ba_r1-2;r1_d!po	2014	0	0	0	0	0	0	0	0	0	2015	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	1
dati;clkoff	2015	0	0	0	0	0	0	0	0	0	2016	3	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
r4_unibus	2016	0	0	0	0	0	0	0	0	0	2017	2	0	3	0	0	0	0	0	0	0	0	0	1	0	0	1	4
ba;d_r1-2;r1_d!po	2017	0	0	0	0	0	0	0	0	0	2020	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	1
dati;clkoff	2020	0	0	0	0	0	0	0	0	0	2021	3	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
r3_unibus	2021	0	0	0	0	0	0	0	0	0	2022	2	0	3	0	0	0	0	0	0	0	0	0	1	0	0	1	3
ba;d_r1-2;r1_d!po	2022	0	0	0	0	0	0	0	0	0	2023	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	1
dati;clkoff	2023	0	0	0	0	0	0	0	0	0	2024	3	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
r2_unibus	2024	0	0	0	0	0	0	0	0	0	2025	2	0	3	0	0	0	0	0	0	0	0	0	1	0	0	1	2
ba;d_r5!sp<-r5	2025	0	0	0	0	0	0	0	0	0	2026	4	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	5
r6_d;dati;clkoff	2026	0	0	0	0	0	0	0	0	0	2027	3	0	3	0	0	0	1	0	0	0	0	0	2	0	0	1	6
r5_unibus!r5<-(sp	2027	0	0	0	0	0	0	0	0	0	2030	2	0	3	0	0	0	0	0	0	0	0	0	1	0	0	1	5
d_r6+2;r6_d	2030	0	0	0	0	0	0	0	0	0	2031	6	0	3	0	1	0	0	0	0	11	2	17	2	0	0	1	6
ba_r6;dati!rtspc	2031	0	0	0	0	0	0	0	0	0	2032	2	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	6
d_r6+2;r6_d;clkof	2032	0	0	0	0	0	0	0	0	0	2033	7	0	3	0	1	0	0	0	0	11	2	17	2	0	0	1	6
r7_unibus;but16	2033	0	0	0	0	0	0	0	0	0	2034	2	0	3	0	0	0	0	0	0	0	0	0	1	0	16	1	7
goto16	2034	0	0	0	0	0	0	0	0	0	16	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d_r5;dato;clkoff	2035	0	0	0	0	0	0	0	0	0	2036	5	0	0	0	1	0	5	0	0	0	0	0	0	0	0	1	5
d_r7;p3!r5<-r7	2036	0	0	0	0	0	0	0	0	0	2037	6	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	7
r5_d	2037	0	0	0	0	0	0	0	0	0	2040	2	0	3	0	0	0	0	0	0	0	0	0	2	0	0	1	5
r0_d!r0<-r5	2040	0	0	0	0	0	0	0	0	0	2041	2	0	3	0	0	0	0	0	0	0	0	0	2	0	0	1	0
d_r6!r5<-r6	2041	0	0	0	0	0	0	0	0	0	2042	4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	6
r5_d	2042	0	0	0	0	0	0	0	0	0	2043	2	0	3	0	0	0	0	0	0	0	0	0	2	0	0	1	5
d;ba_r6-2;r6_d!pu	2043	0	0	0	0	0	0	0	0	0	2044	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	6
d_r4;dato;clkoff	2044	0	0	0	0	0	0	0	0	0	2045	5	0	0	0	1	0	5	0	0	0	0	0	0	0	0	1	4
d;ba_r6-2;r6_d!pu	2045	0	0	0	0	0	0	0	0	0	2046	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	6
d_r3;dato;clkoff	2046	0	0	0	0	0	0	0	0	0	2047	5	0	0	0	1	0	5	0	0	0	0	0	0	0	0	1	3
d;ba_r6-2;r6_d!pu	2047	0	0	0	0	0	0	0	0	0	2050	6	0	3	0	1	1	0	10	0	6	2	17	2	0	0	1	6
d_r2;dato;clkoff	2050	0	0	0	0	0	0	0	0	0	2051	5	0	0	0	1	0	5	0	0	0	0	0	0	0	0	1	2
d_r6-2;r6_d!r6<-r	2051	0	0	0	0	0	0	0	0	0	2052	6	0	3	0	1	0	0	10	0	6	2	17	2	0	0	1	6
d_r0!r7<-r0	2052	0	0	0	0	0	0	0	0	0	2053	4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
r7_d;but16	2053	0	0	0	0	0	0	0	0	0	2054	2	0	3	0	0	0	0	0	0	0	0	0	2	0	16	1	7
goto16	2054	0	0	0	0	0	0	0	0	0	16	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

labels and addresses
==========

bes 2000

Figure 8: fastc.lst

and user-defined RAM tables (cf.subsection 2.5.7), before RAM locations are assigned to the rest of the microinstructions.

<name>.bin

This file is the binary version of the assembled microcode which is loaded into the writable control store. Fig. 9 depicts the file fastc.bin.

```
0000000 000407 001662 000000 000000 000154 000000 000000 000000
0000020 134000 134670 000000 000000 000000 000000 000000 000000
0000040 000000
0000760 000000 000000 000000 000000 000000 020000 020540 060615
0001000 000000 000000 040000 002376 000000 100000 000000 141400
0001020 006371 000210 000033 003000 100410 002366 000000 100026
0001040 003057 146610 002342 000000 000000 000000 040000 000227
0001060 000000 000025 000000 100400 002364 000000 000033 003000
0001100 100410 002370 000000 005000 000000 040000 002367 000000
0001120 100000 000000 141400 006375 000211 005000 000000 040000
0001140 002365 000000 000000 000000 040000 002373 000000 100021
0001160 000000 046000 002363 000000 100021 003057 146610 002362
0001200 000000 000000 000000 060020 002361 000000 040024 000000
0001220 046000 002360 000000 100021 003057 146610 002357 000000
0001240 000000 000000 060020 002356 000000 040023 000000 046000
0001260 002355 000000 100021 003057 146610 002354 000000 000000
0001300 000000 060020 002353 000000 040022 000000 046000 002352
0001320 000000 020025 000000 100600 002351 000000 100026 000000
0001340 066020 002350 000000 040025 000000 046000 002347 000000
0001360 100026 004457 146400 002346 000000 020026 000000 040220
0001400 002345 000000 100026 004457 166400 002344 000000 047027
0001420 000000 046000 002343 000000 000000 000000 040000 000361
0001440 000000 000025 000000 120520 002341 000000 000027 000000
0001460 140400 002340 000000 100025 000000 046000 002337 000000
0001500 100020 000000 046000 002336 000000 000026 000000 100400
0001520 002335 000000 100025 000000 046000 002334 000000 100026
0001540 003057 146610 002333 000000 000024 000000 120520 002332
0001560 000000 100026 003057 146610 002331 000000 000023 000000
0001600 120520 002330 000000 100026 003057 146610 002327 000000
0001620 000022 000000 120520 002326 000000 100026 003057 146410
0001640 002325 000000 000020 000000 100400 002324 000000 107027
0001660 000000 046000 002323 000000 000000 000000 040000 000361
0001700 000000
0003560 000000 000000 060562 046155 051501 000124 000001 020540
0003600 060562 043155 051522 000124 000001 020000 060562 057555
0003620 000143 000000 000001 134000 057557 071146 072163 000000
0003640 000001 000000 057557 060554 072163 000000 000001 000540
0003660 062555 063155 071562 000164 000001 134000 062555 066155
0003700 071541 000164 000001 134670 064143 061545 071553 066565
0003720 000001 060615 041525 052123 052122 000000 000002 000760
0003740
```

Figure 9: fastc.bin

<name>.tab

This file is generated specifically for the microsimulator [5]. It contains information used by the simulator for interperting simulator commands. This information includes field names, field bounds, lower and upper limits (control store addresses) of the microcode generated, register names, macro names, labels, etc. From this file, the simulator generates a symbol table which is used to interpret simulator commands. As a result, simulator commands can contain any symbol string that is recognized by the microassembler, including names of user-defined macros.

4.3 MICRO/40 Error Messages

The errors detected by MICRO/40 may be grouped into three classes.

Non-continuable errors terminate the assembly process. When a non-continuable error is encountered, control is transferred to the UNIX operating system, without generating any object microcode.

Continuable errors cause the microassembler to skip (i.e., no object microcode is generated) the erroneous input line and to continue assembly with the next input line. No object microcode file is generated.

Recoverable errors cause the microassembler to issue a warning. MICRO/40 can by itself recover from these errors and may either skip (i.e., no object microcode is generated) the erroneous input line or assemble it, after the erroneous line has been modified according to internal default mechanisms.

For errors of each class, MICRO/40 issues error message which generally have the following format:

<file name> line <line number>
<source code line>
<message> ,

where
<filename> is the name of the source file being assembled,
<line number> is the line number of the erroneous line in the source file,
<source code line> is the erroneous source code line,
<message> is the error message (the different error messages are disucssed in the following subsections).

Example: /uprog/rtll/vector.mic line 96
d_b; n.z.v.c; skipzero; goto vec5
undefined symbol goto

4.3.1 Non-continuable Errors

For non-continuable errors we may distinguish between internal errors, user errors, system errors, and resource errors.

Internal Errors

Internal errors are fatal errors that are caused by the execution of the microassembler program.

1. "internal error -- duplicate symbol <token>"
   A symbol, <token>, occurs twice in the symbol table.

2. "pop:stack underflow"
   The microassembler attempts to pop an empty stack.

3. "duplicate entry: <token>"
   The microassembler attempts to enter the same symbol, <token>, twice into the symbol table (this message is printed without a header).

4. "internal error -- default case index to pick type <symbol>"
   A non-existing symbol type is found for the symbol, <symbol>.

5. "internal error -- compiling a macrotype"
   A macro name is found after completion of the macro expansion.

6. "internal error -- illegal reference type"
   A symbol of type reference (<emit field>, cf. subsection 2.3.2) is found before symbols of this type were entered into the symbol table.

7. "internal error -- default pseudo type <symbol>"
   The symbol, <symbol>, that is recognized as type pseudo is not a reserved pseudo operator (cf. subsection 2.5).

8. "internal error -- default operand type left <symbol>"
   The symbol, <symbol>, that is found on the left hand side of an assignment statement is not a defined operand (cf. subsection 2.3.1).

9. "internal error -- default type operand right <symbol>"
   The symbol, <symbol>, that is found on the right hand side of an assignment statement is not a defined operand (cf. subsection 2.3.1).

10. "internal error -- default type operator <token>"
    The symbol, <token>, that is recognized as an operator is not a defined operator (cf. subsection 2.3.2).

11. "internal error -- double _ not detected"
    A double occurrence of the assignment operator, _, has not been detected in the first syntax check.

12. "internal error -- bad type in label resolution"
    A symbol of a type other than label or table is found in the control store address resolution.

13. "internal error - bad reference type"
    When relocating microcode, a symbol is found among the control store refer-

ence that is not of type label or table (this message is printed without a header).

User Errors

The following error messages refer to fatal syntax or microprogramming errors.

14. "sourcefile??"
No source file was specified in the command, mic{<opt>}<name>.mic (this message is printed without a header).

15. "illegal file name <name>"
The file name in the command, mic{<opt>}<name>.mic, is not accepted. For example, the suffix, mic, is missing (this message is printed without a header).

16. "rekursive macro call for <macro name>"
Recursive macro calls are not allowed (cf. subsection 2.5).

17. "starting file with open compound"
The require statement (cf. subsection 2.6.3) occurs in a set...tes group. group.

18. "starting file with open set"
The require statement (cf. subsection 2.6.3) occurs in a set...tes group.

19. "ending tile with open compound"
The end-of-file mark of the text file being processed is found before the closing 'end' of a start...end group.

20. "ending file with open set"
The end-of-file mark of the text file being processed is found before the closing 'tes' of a set...tes group.

21. "start within an open compound"
The pseudo operator, start, is found outside a set...tes group or inside another start...end group.

22. "end doesn't balance set"
The pseudo operator, end, is found inside a set...tes group and has no associated 'start'.

23. "set within an open case"
The pseudo operator, set, is found inside a set...tes group and is not inside

24. "tes doesn't balance start"
    The pseudo operator, tes, is found in a start...end group and has no associated 'set'.

25. "set !!! in a macro body"
    set...tes groups are not allowed inside macro bodies.

26. "tes !!! in macrobody"
    set...tes groups are not allowed inside macro bodies.

27. "macro declaration in macrobody not allowed"
    A macro declaration is found during macro expansion.

28. "impossible assignment to lowlim you have <number of statements> statements"
    The microprogram cannot be stored in the specified control store address space (this message is printed without a header).

System Errors

System errors are caused by the inability of the system to execute a given command. The following error messages are printed without a header.

29. "move error -- <file name>"
    An error occured in the generation and storage of the object microcode file, <name>.bin.

30. "cannot find <file name>"
    A microassembler or microsimulator file, <file name>, cannot be found.

31. "sorry ! try again"
    The system is unable to create a new UNIX process.

32. "fatal error in <file name>"
    The file,<file name>, being executed contains a fatal error.

33. "cannot create file: <file name>"

34. "cannot open file: <file name>"

35. "read error on file: <file name>"

36. "write error on file: <file name>"

37. "cannot close file: <file name>"

38. "cannot remove file: <file name>"

39. "error on file: <file name>"

40. "seek error on file: <file name>"

Resource Errors

Resource errors are caused by overflow of MICRO/40 information resources.

41. "symbol table full <symbol>"
The attempt to put the symbol, <symbol>, into the symbol table causes a symbol table overflow. The size of the symbol table is 1024 words.

42. "string table overflow"
The size of the string table is 10,240 characters.

43. "push: stack overflow"
The size of the stack is 100 words.

44. "reference table overflow"
The size of the reference table is 200 words.

45. "all 1024 ram locations are used"
The user microprogram exceeds the capacity of the writable control store.

46. "too much sets"
The maximum number of tes...set groups is 150.

47. "too much tables"
The maximum number of tables is 20(cf. subsection 2.5.7).

48. "no free locations remaining for sets"
It is impossible to allocate control store locations with 2^n address boundaries for the allocation of set...tes groups.

49. "no free location remaining for tables"
It is impossible to allocate control store locations for table declarations.

4.3.2 Continuable Errors

50. "illegal symbol where number expected <number>"
A number, <number>, includes an illegal character.

51. "no)following (<number>"
A negative number, <number>=-XXX, of the form, (-XXX, is found.

52. "redefining a symbol <symbol>"
Macro name or label, <symbol>, is used twice.

53. "illegal use of label <label>"
A label reference, <label>, occurs on the left hand side of an assignment.

54. "illegal use of table <table>"
A table name, <table > , cannot occur on the left hand side of an assignment statement.

55. "<op> operator with no operand"
An operater, <op>, is used without operand.

56. "illegal symbol <symbol>"
An undefined identifier, <symbol>, is used.

57. "undefined symbol <token> "
The symbol, token , following a label definition or a macro definition is unequal ':' or ':='. respectively.

58. "inconsistent use of emit-field"
Assignments to the EMIT field and other microoperation fields in XU<79:64> are made in a single microinstruction.

59. "<field> not followed by ="
'=' is missing in an assignment to the microoperation field, <field>.

60. "illegal field assignment <field>"
An identifier is assigned to a microoperation field, <field>, other than EMIT or xupf.

61. "missing] in table reference"
Syntax: <table name>[<table index>]

62. "code on same line as finis"

63. "code on same line as start"

64. "code on same line as end"

65. "no code between start - end"

66. "code on same line as set"

67. "code on same line as tes"

68. "set following a start not allowed"
The pseudo operator, set, must not immediately follow the pseudo operator, start (cf. subsection 2.5.10).

69. "set size less than 2 not allowed"
(cf. subsection 2.5.9)

70. "code on same line as lowlim"

71. "lowlim must be first statement"
The pseudo operator, lowlim, must be the first statement in a source file (if used).

72. "missing = in lowlim"
Syntax: lowlim = <value>

73. "illegal address to lowlim"
<value> in,<lowlim>= <value>, must be in the range [2000:3777].

74. "illegal operation to'.'"
Syntax: . = <value> (cf. subsection 2.5.4).

75. "illegal absolute address assignment"
<value> in,. = <value>, must be in the range [2000:3777].

76. "cannot assign absolute address to controlled words in a set"
The micorassembler allocates control store locations for all micro-instructions in a set...tes group.

77. "absolute location assigned twice"
The same absolute control store address is assigned to two different microinstructions.

78. "code on same line as <table>"
Table declarations and preloads must occur on a line by themselves (cf. subsections 2.5.7 and 2.5.8).

79. "illegal table name <table>"
An illegal idenfier is used as name for the table, <table>.

80. "duplicate use of <table>"
The identifier used as name of the table, <table>, is used twice.

81. "no go to before table declaration"
Table declarations must be preceeded by an explicit assignment to the xupf field.

82. "arithmetic expression too long"
An expression can in maximum contain 15 operands and operators.

83. "registers can only be read as full words"
The selectors, 1 and h, cannot be used in occurrences of general purpose registers on the right hand side of assignment statements (cf. subsection 2.3.1).

84. "multiple _ in assignment statement"

85. "illegal assignment to eubc"
Only data at the output of the shift/mask unit can be assigned to eubc. Simultaneous gating of the shift/mask unit output onto the EUBC BUS and the RD BUS are not allowed.

86. "impossible assignment on dmux or to stack"
(cf. subsection 2.3)

87. "impossible assignment to ba or d register"
(cf. subsection 2.3)

88. "two different alu functions chosen"
Only a single ALU function can be specified per microinstruction.

89. "no such alu function <function>"
(cf. subsection 2.3.2)

90. "unrecognizable statement"
For example, two consective ';'.

91. "undefined symbol <symbol>"
For example, undefined carrier on the left hand side of an assignment statement.

92. "illegal element in this context <token>"
For example, an identifier occurs, where an operator is expected.

93. "illegal assignment to stackpointer"
Stack pointer values must be in the range [0:15].

94. "missing ; after sp assignment"

95. "illegal assignment to <operand>"
(cf. subsection 2.3.2)

96. "impossible assignment statement"
For example, operand missing (cf. subsection 2.3.2).

97. "illegal use of ba, ir, sf, df or eubc"
These identifiers cannot occur on the right hand side of an assignment statement.

98. "illegal type in assignment statement"
An assignment statement contains a token of illegal type (cf. subsection 2.3.2).

99. "illegal use of ^"
'^' can only be used with extension operands (cf. subsection 2.3.2).

100. "illegal use of /"
'/' can only be used in the identifier d/2.

101. "d,dshift,ba,ir,sf,df,eubc, or unibus are not allowed as source to expression"
The specified carriers must not occur in a compound expression (cf. subsection 2.3.2).

102. "illegal type in arithmetic expression"
An arithmetic expression contains a token of illegal type (cf. subsection 2.3.2).

103. "register specification with no ["
Syntax: R[<index>]

104. "illegal general register specification <token>"
<index>::= 0|1|2|3|4|5|6|7|10|11|12|13|14|15|16|17|8.|9.|10.|11.|12.|
13.|14.|15.|ba|df|sf|

105. "no] after register specification"
syntax :R[<index>]

106. "illegal register modifier <modifier>"
A register selector other than '1' or 'h' is used (cf. subsection 2.3.1).

107. "no > after register modifier"
Syntax: R[<index>]{<<selector>>}

108. "inconsistant general register specification"
Two different general purpose registers are used in the same microinstruction.

109. "illegal sbmh modification"
A <high selector> other than h, e, 1, or c is used for the B Register. (cf. subsection 2.3.1)

110. "illegal sbml modification"
A <low selector> other than h, z, 1, or c is used for the B Register (cf. subsection 2.3.1).

111. "no > after sbm modification"
Syntax: b{<<B modifier>>}

112. "b - constant with no ["
Syntax: c[<number>]

113. "illegal b - constant"
<number> in, c[<number>] must be in the range [0:15].

114. "no] after b - constant"
Syntax: c[<number>]

115. "ram with no ["
Syntax: RAM [<specifier>]

116. "illegal ram address field <specifier>
<specifier>in, RAM [<specifier>] must be s or tos.

117. "no] after ram"
Syntax: RAM[<specifier>]

118. "field selection with no >"
Syntax: <extension carrier>{<<field selection>>}{<shift><number}

119. "impossible field selection"
(cf. subsection 2.3.2)

120. "no code to continue after last set of program"
Every set...tes group must be succeeded by at least one microinstruction in the source microcode (cf. subsection 2.5.9).

121. "undefined label"

4.3.3 Recoverable Errors

122. "flag <opt>??
The assembly flag, <opt>, in the microassembler call, mic{<opt>}<name>.mic, is not recognized (cf. subsection 4.1).

123. "only one filename allowed <file name> will be ignored"
The second file, <file name>, specified in the microassembler call, mic {opt}<name>.mic, is ignored.

124. "truncated from line <line number>"
The input line contains too many characters and is truncated after line <line number>. The maximum number of characters in a MICRO/40 input line is 300.

The error messages for the following recoverable errors are preceeded by the word, warning, on a separate line.

125. "no $ following macro <macro>"
The definition of macro, <macro>, is not terminated by '$'.

126. "no finis found"
(cf. subsection 2.5.3)

127. "illegal dest/msc function -- s assumed"
The PPE field is not set and the DEST/MSC fields specify a push or pop operation. One of the following DEST/MSC functions is assumed, according to the context of the given microinstruction:
push; RD ← emit; s ← DMUX,
push; eubc ← emit; stack ← DMUX,
RD ← emit; RAM[s] ← DMUX; pop,
eubc ← emit; RAM[s] ← DMUX; pop.

128. "This line generates dest/msc function 'off'"
The data paths and functional units in the WCS 11/40 are not used.

129. "you are changing the value of field <field name>"
The input line contains two assignments to the microoperation field, <field name>. The right most field assignment is used.

130. "field value too large for <field name>"
The value assigned to <field name> is not in the range of legal field values. The value is truncated.

131. "less than 2 exp(n) statements in this set"
Control store locations with 2^n address boundaries are allocated for the microinstruction in a set...tes group.

5. Terminal Session

In this section, we demonstrate the operation of the MICRO/40 assembler by a commented protocol of a terminal session. To this end, several errors have artificially been introduced into the example microprogram, fastc.mic (cf. Fig.7). These errors are corrected using the UNIX text editor [2], and the corrected microprogram is assembled. System commands and responses start at the left margin of the page. Comments are indented. Responses from the UNIX operating system end with the prompt '%'.

```
% mic -a fastc
```
An attempt is made to assemble a file whose name does not have the suffic, mic.
```
illegal filename fastc
```
Non-continuable error.
```
% mic -a fastc.mic
```

```
          MICRO/40 is called with a correct filename.
fastc.mic
defs.mic
26 lines read.
     The microassembler acknowledges the acceptance of the 'required' file,
     defs.mic, which contains 26 lines.
fastc.mic line 4
.=2001; d_210; b_d
illegal symbol where number expected 210
     Continuable error.
fastc.mic line 11
set
set within an open case
     Non-continuable error.  Control is returned to UNIX.
%  ed fastc.mic
     The UNIX test editor is called with the file,
     fastc.mic, as an argument.
905
     The editor lists the number of characters in the file fastc.mic.
11
     Line 11 of fastc.mic is requested.
set  ! check for other instr
     Line 11 of fastc.mic is listed.
s/set/start/p
     Line 11 of fastc.mic is edited.
start  ! check for other instr.
     The corrected line 11 of fastc.mic is listed.
4
.=2001; d_21o; b_d
s/o/0/p
.=2001; d_210; b_d
     Line 4 of fastc.mic is corrected.
w
     The corrected lines are written back into fastc.mic.
907
     The new number of characters in fastc.mic is listed.
q
     Editing is terminated.  Control is returned to the operating system.
%  mic -a fastc.mic
     MICRO/40 is called again.
```

```
fast.mic
defs.mic
26 lines read
     The acceptance of the file, defs.mic, is acknowledged.
fastc.mic line 24
doti; clkoff
undefined symbol doti
     Continuable error.
fastc.mic line 25
r4__unibus
multiple _ in assignment statement
     Continuable error.
warning
fastc.mic line 28
r7_unibus; but 16; ubf=12
you are changing the value of field ubf
     Recoverable error.
66 lines read
     MICRO/40 acknowledges the acceptance of the file fastc.mic.  Control is
     returned to UNIX.
%  ls
     A listing of the names of the existing files is requested.
defs.mic
fastc.mic
     No object microcode has been generated.
%  ed fastc.mic
907
24
doti; clkoff
s/o/a/p
dati; ckoff
+
     Request for next (25) line of fastc.mic .
r4__unibus
s/--##__/_/p
r4_unibus
38
r7_unibus; but 16; ubf=12
```

```
s/ubf=12//p
?
```

The editor asks for an acknowledgement of the deletion.

```
s/ubf=12//p
r7_unibus; but 16;
s/6;/6/p
r7_unibus; but 16
w
899
q
```

Correction of lines 24, 25, and 38 of fastc.mic.

```
%  mic -a fastc.mic
```

MICRO/40 is called again.

```
fastc.mic
defs.mic
26 lines read
66 lines read
```

No errors are detected and the file, fastc.mic is assembled. Control is returned to UNIX.

```
%  ls
```

A listing of the names of the existing files is requested.

```
defs.mic
fastc.ass
fastc.lst
fastc.mic
fastc.tab
```

The files, fastc.ass, fast.lst, and fastc.tab, have been generated, as MICRO/40 was called with the assembly flag, -a.

```
%  as fastc.ass
```

The UNIX assembler is called to generate the object microcode.

```
%  mv a.out fastc.bin
```

The generated file is stored in the file names fastc.bin. Note that fastc.bin is directly generated by MICRO/40, if no assembly flag or the assembly flag, -s, is used in the microassembler call.

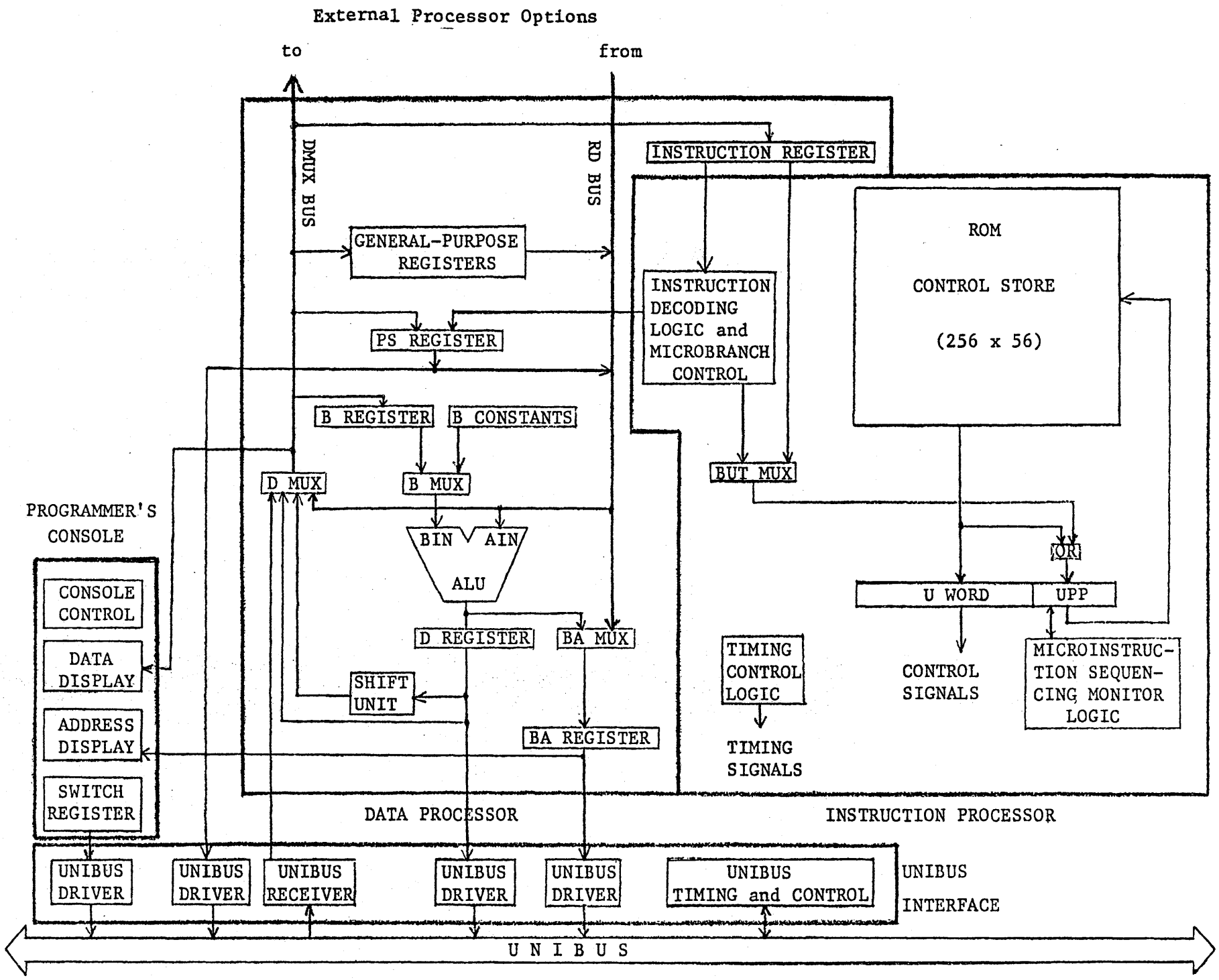
Acknowledgement

The MICRO/40 assembler was originally implemented by R. Kallerhoff of the Technical University Berlin. The authors are indebted to H. Mauersberg,

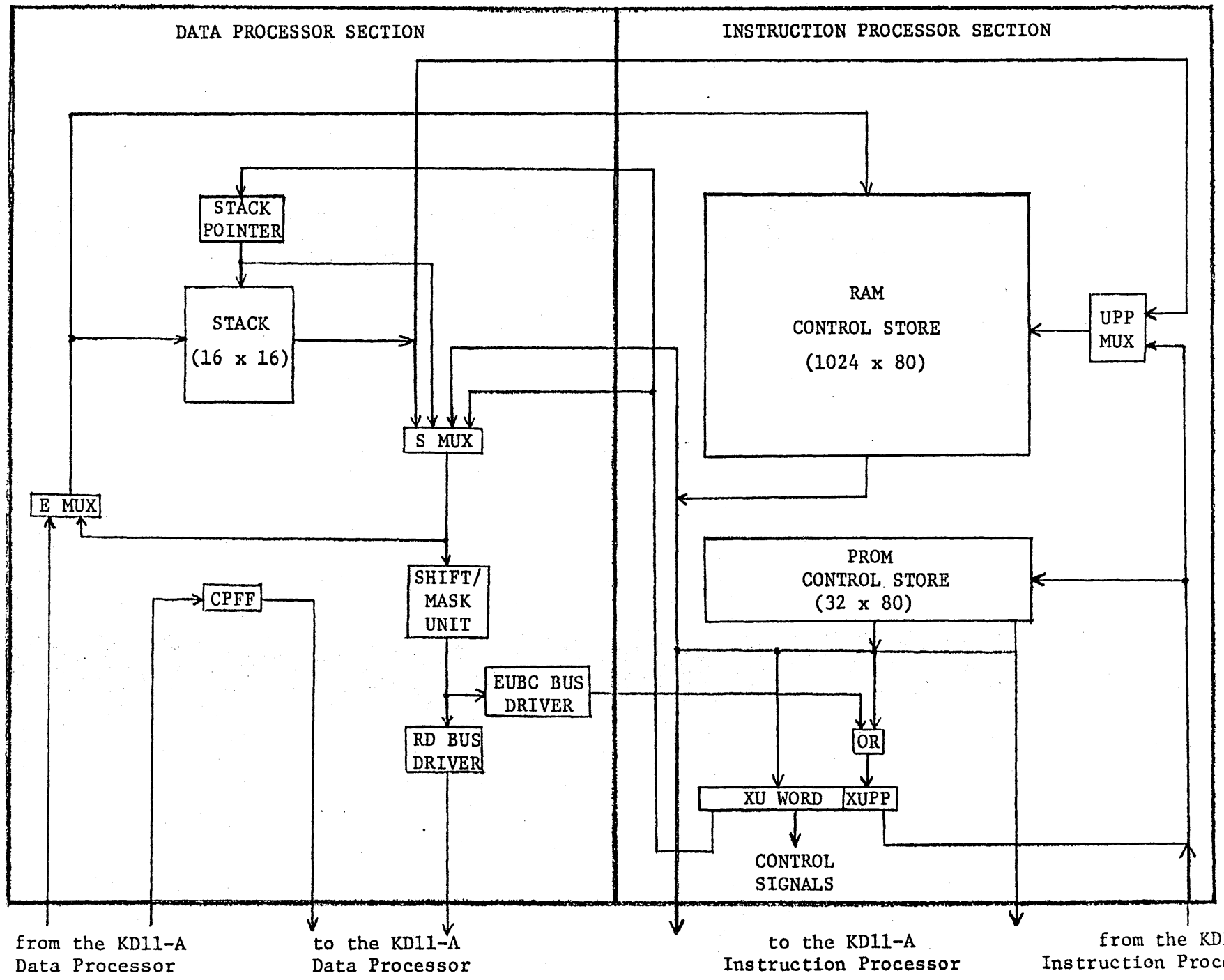
also with the Technical University Berlin, for providing us with the micro-assembler and for many helpful suggestions concerning the development of a microprogramming laboratory around a PDP-11/40E. Thanks are also due to B. E. Blasing who helped with the installment of the microassembler in our microprogramming laboratory. The authors wish to express their gratitude to Professors W. K. Giloi and W. R. Franta for initiating the microprogramming laboratory project at the University of Minnesota. The microprogramming laboratory is funded by University Computer Services, University of Minnesota.

Appendix

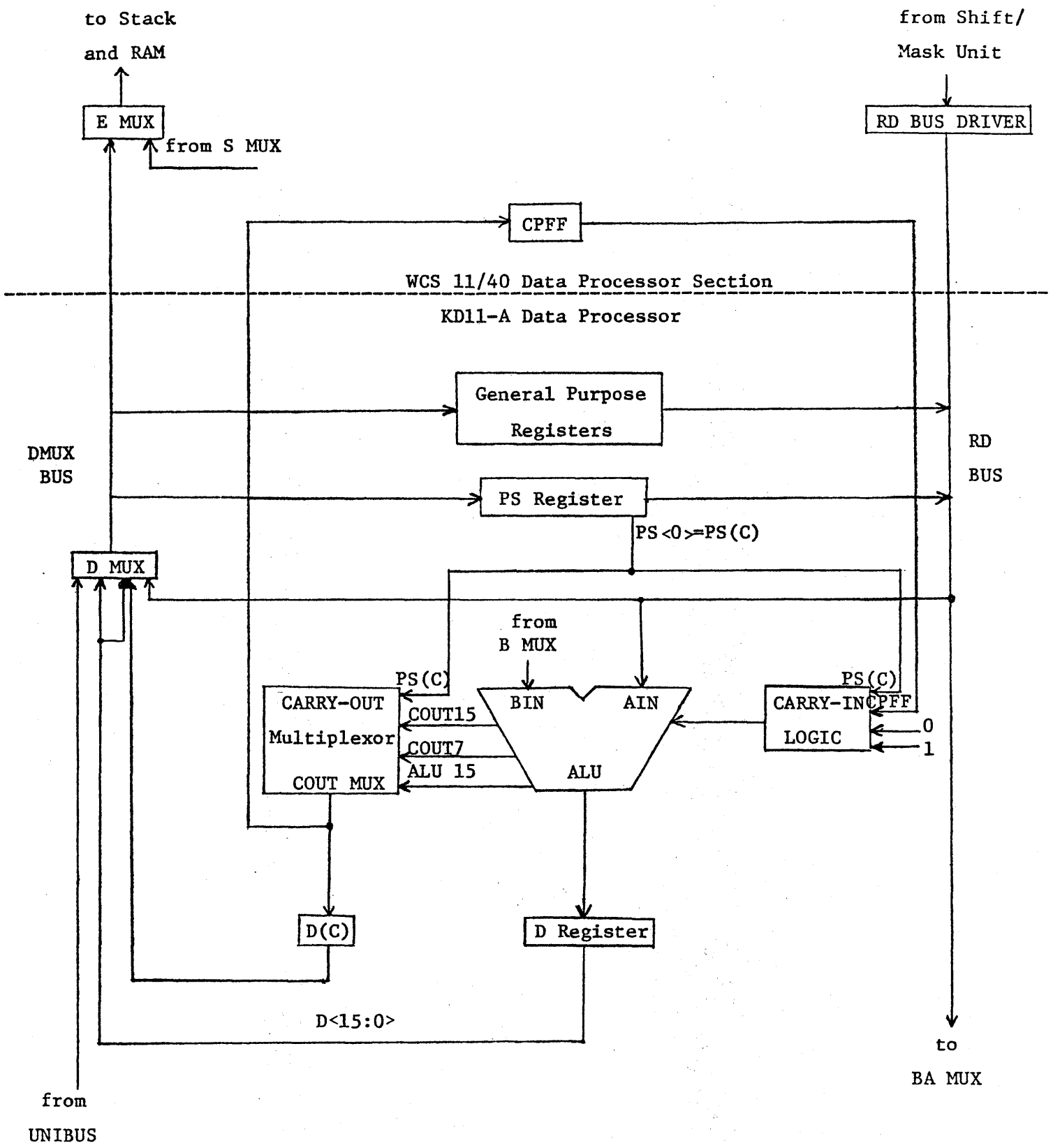
PDP-11/40E Register-Transfer Block Diagrams



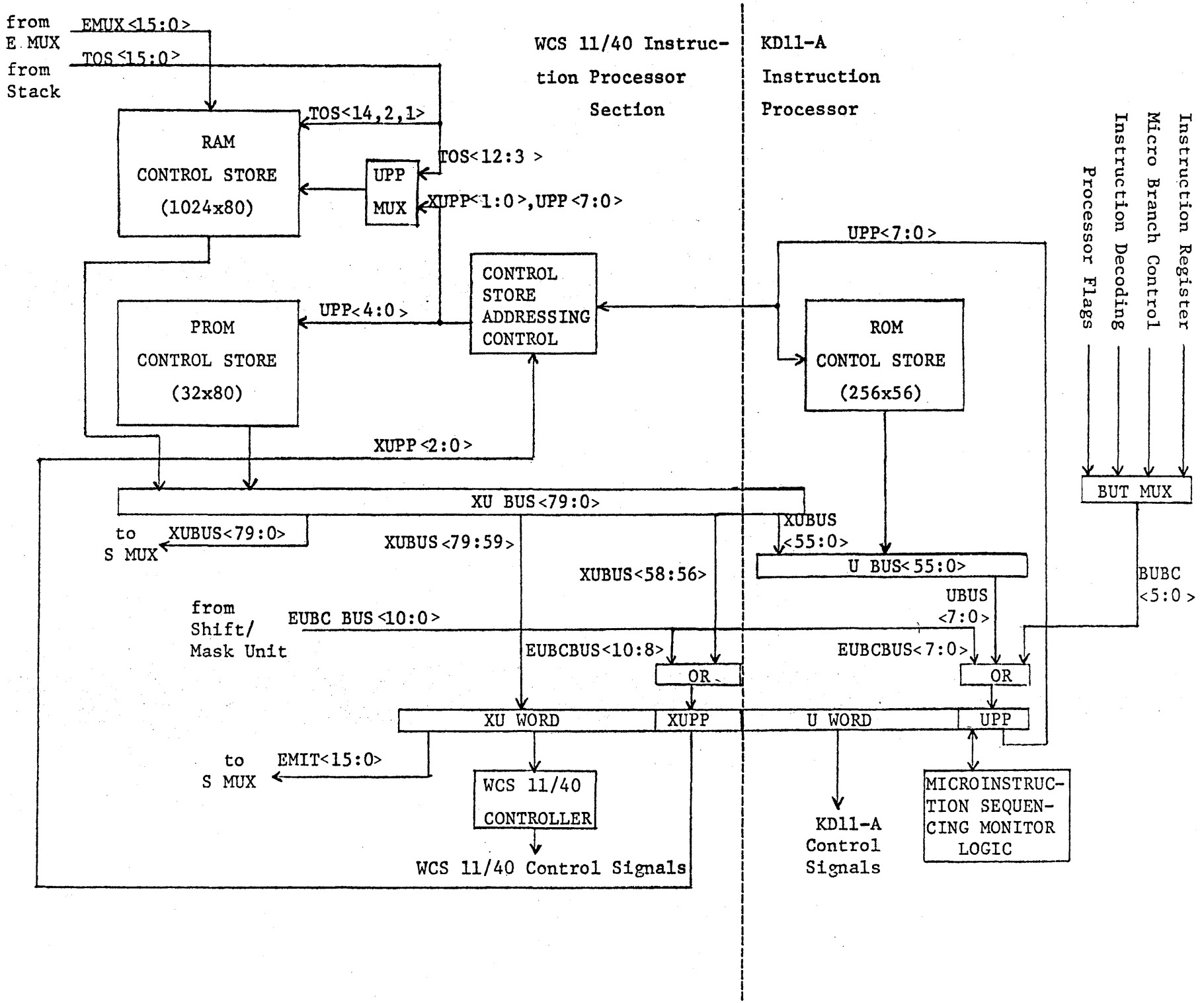
KD11-A Processor Register-Transfer Diagram



WCS 11/40 Register Transfer Diagram



Data Processor Interface



Instruction Processor Interface

References

[1] Fuller, S. H.; Almes, G. T.; Broadley, W. H.; Forgy, C. L.; Karlton, P. L.; Lesser, V. R.; Teter, J. R., "PDP-11/40E Microprogramming Regerence Manual," Department of Computer Scienc, Carnegie-Mellon University, Tech. Report 16-January-76.

[2] UNIX Documentation Book I, "Introduction to UNIX," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[3] UNIX Documentation Book II, "UNIX Programmer's Manual Section I - Commands," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[4] UNIX Documentation Book III, "The "C" Programming Language," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[5] Berg, H. K.; Blasing, B. E., "PDP-11/40E Microcode Simulator Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-10.

[6] Mueller, J., "SMILE - Manual," Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Betriebssysteme, Technical University Berlin, Dec. 1976.

[7] Berg, H. K.; Samari Kermani, N., "A Primer on the SMILE Microprogram Load and Test System," Department of Computer Science, University of Minnesota, Tech. Report 78-11.

[8] Teter, J. R., "PDP-11/40E Hardware Maintenance Manual," Department of Computer Science, Carnegie-Mellon University, September 1976, revised June 1977.

[9] Berg, H. K., "A PDP-11/40E Microprogramming Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-8.

[10] Berg, H. K.; Covey, C. R., "A Primer on the Use of a Logic State Analyzer as a Microprogram Debugging Aid," Department of Computer Science, University of Minnesota, Tech. Report 78-12.