SOFTWARE MICROPROGRAMMING TOOLS FOR THE VAX-11/780

by

Richard D. Tuck

Memorandum No. UCB/ERL M79/65

September 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720
Research supported in part by the U.S. Department of Energy Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358, and the National Science Foundation Grant MCS-78-7291.

Partial fulfillment of Master of Science in Engineering.
CHAPTER 1

INTRODUCTION

Although the concept of microprogramming is nearly as old as electronic computing [Wilkes 51], it has only gained widespread popularity as an implementation technique since the introduction, in the early 1960's, of the IBM-360 family of computers. When a microprogram is used strictly as an implementation tool, the programmer is usually very familiar with the microarchitecture, if not its designer. Lately, however, microprogramming has been used for more than an instruction-set implementation tool: more and more operating systems routines are being microcoded [Stockenberg 78][Bondy 77], and several machines are user microprogrammable [Agrawala 78]. Also, as machine architectures become more complex, the supporting microprograms grow larger; the IBM-360, model 50 (1965) control store had a capacity of 2,818, ninety-bit words [Husson 70]; the VAX-11/780 (1978) control store has a capacity of (at least) 5,120, ninety-six-bit words. Larger microprogramming teams are required just to implement the instruction sets. Due to these factors, the people writing microprograms are less often the designers of the microarchitectures with which they are working, and are correspondingly less familiar with them.

The traditional microprogramming tools have been rather crude - either a flow-chart language, as used for the IBM-360s [Husson 70], or a more traditional assembly language [Davidson 78]. These sufficed for expert microprogrammers and compact microprograms, but are no longer any more appropriate than is assembly language for all systems and applications
macro-programming.

The desideratum, then, is a High Level Systems Program Language (HL-SPL) which could be translated into an efficient microprogram. As a first step towards this, we have designed and implemented a low-level language which hides from the programmer most of the baroque features of a particular microarchitecture. This language (Yet Another Low-Level Language, or YALLL) can be used as the machine-independent intermediate code output of a HL-SPL microcode compiler [Patterson 79]. The key observation concerning microarchitectures is not their differences, but their similarities: microinstructions invariably transform data between registers, and reference memory only in loading and storing their contents.

This paper describes the implementation of the YALLL microprogramming system running under Unix on Digital Equipment Corporation's VAX-11/780 [Strecker 78], hereafter known as YALLL/VAX. The VAX was chosen for this work because it is a new computer with 1024 words of user-programmable control store. Also, there are no other facilities for microprogramming this machine, as of this writing; those announced by the manufacturer will run under the VMX operating system, not Unix. Finally, it is anticipated that the VAX-11/780 will be extremely hard to microprogram even using DEC's announced macro-microassembler, because of the great microinstruction width.

The main constituent of this system is the VAX/YALLL compiler, called yc; this is a program consisting of 4456 lines of C [Ritchie 78], 495 of YACC [Johnson 78], and 283 of Lex [Lesk 75]. The YACC and Lex programs translate into 876 and 1404 lines of C, respectively. This translator contains an assembler mode embedded in the language. It is the only microassembler
available for the machine at this time. The YALLL/VAX language, the assembly mode, and the use of yc are described in chapter three. The internals of the compiler are outlined in Appendix II, written mainly for the compiler maintainer.

The other software microprogramming tools are also described in chapter three: a linker, symbolic dump programs, floppy disk file transfer program, and DEC'S console program and console microdebugger. This description includes a tutorial example.

The approach described here, and supported by these programs, requires that the microprogram be written on the console floppy disk and loaded into writable control store (WCS) using a console LSI-11 command. The alternative to this is to load WCS directly from the VAX using the privileged WTPR instruction. The disadvantage of the latter approach is that failing to write a WCS image on the floppy disk does not permit the use of the console microdebugger to its fullest advantage. In particular, examining and changing WCS locations is impossible without the image file. These facilities may make little difference to the YALLL source programmer, but their absence would make microassembly-level debugging much more tedious than necessary.

A large part of this report is concerned with the VAX-11/780 architecture and microarchitecture. Chapter two presents a brief overview, and Appendix I is a fairly thorough description of the macro-architecture, the microarchitecture, and how they relate.
CHAPTER 2

VAX Architecture Summary

This chapter presents a quick overview of the architecture of the VAX, and of the microarchitecture supporting it. Familiarity with this material will aid the understanding of that which follows, especially the examples of chapter four. The architectural features presented here are not necessarily the most interesting aspects of the machine, but those which most profoundly affect the microarchitecture, and thus the writing of microcode. For example, the variable length of the instructions affects how the microcode interacts with the instruction-fetch unit, and the size of data types supported determines how memory is accessed. All the material here is presented in much greater detail in Appendix A; in particular, the microarchitecture and microinstruction fields are presented there in great detail.

1. MACRO ARCHITECTURE

The architecture of the VAX-11 is based on that of the successful PDP-11 family of computers. The VAX does have a "compatibility mode", for running PDP-11 programs, but when running native-mode instructions, only "cultural compatibility" is maintained. That is, VAX data types are similar to corresponding PDP-11 data types. The VAX instruction format is also similar to that of the PDP-11; an opcode, followed by operand-specifying bits, using one of several address modes. VAX instructions may have zero to six operands, whose specifiers vary in length from one to nine bytes, as detailed in section 1.1.2.2 of Appendix A. Data types supported are: binary, two's complement integers (of length one, two, or four bytes), floating point numbers
(four or eight bytes), packed decimal strings (to thirty-one digits), and character strings. The architecture is basically register oriented, but some of the addressing modes facilitate the use of a stack, both for expression evaluation and local variables. There are sixteen, thirty-two-bit "general" registers, one of which (register r15) is the program counter, pc. Other registers are used by special instructions or convention including the stack pointer, sp (r14), frame pointer, fp (r13), and argument pointer, ap (r12). Furthermore, the decimal and character string instructions use some of registers r0–r5.

The VAX-11/780 has a $2^{32}$ byte address space, of which the top fourth is "reserved" and unusable. The remaining addresses are divided into three regions: P0, P1, and system. Logical addresses are translated to physical addresses by way of paging. Each region has its own page table, containing one entry for each 512-byte logical page in that region.

The physical memory, as well as peripheral devices, is connected to the CPU by way of a hierarchy of busses, the principle one of which is the Synchronous Backplane Interconnection (SBI). The SBI has a $2^{28}$ longword address space, encompassing memory, secondary storage devices, and I/O devices.

2. MICROARCHITECTURE

The heart of the VAX CPU is the 32-bit ALU and its associated registers. A simplified data path diagram is given in Figure 1. The general registers (r0–r14, but not pc) are kept in a register file (RAB). One set of registers' output passes through latch LA to the A (right) side of the ALU; the other passes through latch LB to its B (left) side. A file of temporaries (RC) is available on the ALU's left, passing through latch LC. A set of 64 16-bit constants
is also available on the ALU's left. Two very important registers, D and Q can be gated to either side of the ALU—they have several special properties, especially in regards to shifting. The 32-bit D register acts as the memory data register—all data routed to or from memory must pass through it. The "internal data" or ID bus takes its data from D and delivers to Q. This bus connects to several control registers (such as the alternate stack pointers, page table origin and length registers), as well as the instruction buffer (fetch-ahead unit). Instruction-stream data is received by this path, as are branch displacements.

The rotation unit takes the 64 bits from the Q and D registers (Q on the left), rotates by the amount specified by the contents of SC (or another source), and deposits 32 bits of the result back in D. A positive count denotes left rotation while a negative count denotes right rotation. SHF is also a limited shifter used for scaling index values.

The memory address register VA, and instruction buffer address IBA, can be loaded from the ALU output; which of them is used as a memory reference address depends on the destination of the data—VA is used for data fetches (via register D) and IBA for program stream fetches (to the instruction buffer). Either of these registers may be loaded into the PC, which may be incremented using a dedicated adder, thereby avoiding use of the main ALU. In order to speed the handling of floating point quantities in machines without the optional floating point accelerator, an auxiliary, 10-bit ALU (called EALU) is provided. The major component in its data paths is the SC register, which is also used for shifting operations. Other registers associated with it are FE and STATE.
**VAX microinstruction format.** Control words are 96 bits wide, and divided into 30 fields (see Figure 2). Because of the great control word width, considerable parallelism is possible. Thirteen bits of each microinstruction are used to form the address of the successor instruction. When straight-line microcode is being executed, this address is used directly. But, when any conditional branches are taken (governed by the BEN microword field), other information is also used. BEN selects one of twenty-six groups of three, four, or five condition bits. For example, group "1A" is the PSL condition code bits: N, Z, V, & C. These conditions are ORed with the low-order bits of the microaddress field (JMP) to form the address of the successor microword. If the SUB field is one, a microsubroutine call is specified, and the address of the current microword is pushed onto a (sixteen deep) stack before the branch is taken. If the SUB field is a two, an address is popped off this stack, and is ORed with the instruction's JMP field, as well as any conditions specified, to form the next word address.
<table>
<thead>
<tr>
<th>EALU</th>
<th>JMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>13 12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IEK</th>
<th>MSC</th>
<th>V</th>
<th>E</th>
<th>S</th>
<th>K</th>
<th>K</th>
<th>K</th>
<th>CCK</th>
<th>EBMX</th>
<th>SMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>20</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>MCT/CID</th>
<th>FS</th>
<th>SPO</th>
<th>PCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>46</td>
<td>43</td>
<td>42</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KMX</th>
<th>SI/ACM</th>
<th>QK</th>
<th>SGN</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>58</td>
<td>57</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>54</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td></td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DT</th>
<th>X</th>
<th>BEN</th>
<th>ACF</th>
<th>ALU</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>78</td>
<td>77</td>
<td>76</td>
<td>72</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>66</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IBC</th>
<th>DK</th>
<th>SHF</th>
<th>BMX</th>
<th>AMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>92</td>
<td>91</td>
<td>88</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>85</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>82</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 - VAX microinstruction format
1. YALLL Language Description

However widely computer macro-architectures vary, their supporting microarchitectures are remarkably similar. While macro-instructions may deal with control blocks, stacks, queues, and character strings, the microinstructions are concerned with registers, ALU functions, and transfers of data to and from memory. The YALLL microprogramming language deals with these same sorts of primitives: all arithmetic and logical operations are between registers, and the only accesses of main memory are via loads and stores. Statements are also provided for microprogram sequence control (conditional branch, subroutine call), and to control the binding of variables to registers. YALLL is thus very much like the assembly language for a machine such as the Data General Nova. One YALLL language statement is written on each line, with the exception of the jtab statement, which will be discussed later. The semicolon serves not as a statement separator, but as a comment escape; nothing written between the semicolon and the line's end affects the meaning of the program. A label and a colon may precede a statement, making it the possible destination of a jump. In the absence of branching statements, YALLL statements are executed in the order in which they appear in the source program (though they may not be loaded into control store in this order). Programs may not be self modifying, and there is no explicit means of accessing data in control store.
1.1.
Syntax (diagram)

\[
\begin{align*}
\text{load} & \quad \text{[physical]} \; \text{reg}, \quad \{ \text{reg} \; [\pm \text{cezp}] \} \\
\text{stor} & \quad \{ \text{reg} \; \text{cezp} \} \\
\text{move} & \quad \{ \text{reg} \; \text{cezp} \} \\
\text{cmpl} & \quad \text{reg}, \quad \{ \text{cezp} \} \\
\text{add} & \quad \text{addl} \\
\text{sub} & \quad \text{subl} \\
\text{and} & \quad \text{reg}, \quad \{ \text{reg} \; \text{cezp} \}, \quad \{ \text{reg} \; \text{cezp} \} \quad \{ \text{ence} \} \\
\text{or} & \quad \text{xor} \\
\text{srl} & \quad \text{sra} \\
\text{srl} & \quad \text{src} \\
\text{sla} & \quad \text{sic} \\
\text{slc} & \quad \text{jump} \quad \{ \text{label} \} \quad \{ \text{if} \; \text{reg} \; \text{relop} \; \{ \text{reg} \; \text{cezp} \} \} \\
\text{call} & \quad \{ \text{label} \} \\
\text{rtn} & \quad \text{jtab} \; \text{reg} \; [\text{cezp} \; \text{cezp}] \; \text{of} \\
& \quad \{ \text{cezp} \; [\text{cezp} \; \text{sel}] \; \text{sel} \; \text{label} \}^+ \\
& \quad \{ \text{else label} \} \\
& \quad \text{etab} \\
\text{exit} & \quad \{ \text{reg} \; [\pm \text{cezp}] \} \\
& \quad \{ \text{cezp} \}
\end{align*}
\]
1.2.1. Registers, Types

1.2. Semantics

The YALLL programmer may use the machine's built-in register names to designate transfer operands, or may bind symbolic names to them, and use the latter in a microprogram. This symbolic binding gives the programmer the opportunity to specify types for his variables. The type indicates

\[ \text{number} ::= \left[ \begin{array}{c}
\text{decimal number} \\
\text{octal number} \\
\text{hexadecimal number}
\end{array} \right] \]

<table>
<thead>
<tr>
<th>relop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; )</td>
</tr>
<tr>
<td>( &lt;= )</td>
</tr>
<tr>
<td>( &gt;= )</td>
</tr>
<tr>
<td>( &lt;&gt; )</td>
</tr>
</tbody>
</table>

\[ \text{sel} \]

<table>
<thead>
<tr>
<th>binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[0]}}# )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(X's in don't care positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[XIX]}#} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

\[ \text{cx} \neq \text{cx} \]

\[ \text{cx} \neq \text{cx} \]

\[ \text{d}
\]

\[ \text{cx} \]

\[ \text{name} \]

\[ \text{number} \]

end

begin

1.2. Semantics

The YALLL programmer may use the machine's built-in register names to designate transfer operands, or may bind symbolic names to them, and use the latter in a microprogram. This symbolic binding gives the programmer the opportunity to specify types for his variables. The type indicates

\[ \text{number} ::= \left[ \begin{array}{c}
\text{decimal number} \\
\text{octal number} \\
\text{hexadecimal number}
\end{array} \right] \]

<table>
<thead>
<tr>
<th>relop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; )</td>
</tr>
<tr>
<td>( &lt;= )</td>
</tr>
<tr>
<td>( &gt;= )</td>
</tr>
<tr>
<td>( &lt;&gt; )</td>
</tr>
</tbody>
</table>

\[ \text{sel} \]

<table>
<thead>
<tr>
<th>binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[0]}}# )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(X's in don't care positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[XIX]}#} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

\[ \text{cx} \neq \text{cx} \]

\[ \text{cx} \neq \text{cx} \]

\[ \text{d}
\]

\[ \text{cx} \]

\[ \text{name} \]

\[ \text{number} \]

end

begin

1.2. Semantics

The YALLL programmer may use the machine's built-in register names to designate transfer operands, or may bind symbolic names to them, and use the latter in a microprogram. This symbolic binding gives the programmer the opportunity to specify types for his variables. The type indicates

\[ \text{number} ::= \left[ \begin{array}{c}
\text{decimal number} \\
\text{octal number} \\
\text{hexadecimal number}
\end{array} \right] \]

<table>
<thead>
<tr>
<th>relop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; )</td>
</tr>
<tr>
<td>( &lt;= )</td>
</tr>
<tr>
<td>( &gt;= )</td>
</tr>
<tr>
<td>( &lt;&gt; )</td>
</tr>
</tbody>
</table>

\[ \text{sel} \]

<table>
<thead>
<tr>
<th>binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[0]}}# )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(X's in don't care positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[XIX]}#} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

\[ \text{cx} \neq \text{cx} \]

\[ \text{cx} \neq \text{cx} \]

\[ \text{d}
\]

\[ \text{cx} \]

\[ \text{name} \]

\[ \text{number} \]

end

begin

1.2. Semantics

The YALLL programmer may use the machine's built-in register names to designate transfer operands, or may bind symbolic names to them, and use the latter in a microprogram. This symbolic binding gives the programmer the opportunity to specify types for his variables. The type indicates

\[ \text{number} ::= \left[ \begin{array}{c}
\text{decimal number} \\
\text{octal number} \\
\text{hexadecimal number}
\end{array} \right] \]

<table>
<thead>
<tr>
<th>relop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; )</td>
</tr>
<tr>
<td>( &lt;= )</td>
</tr>
<tr>
<td>( &gt;= )</td>
</tr>
<tr>
<td>( &lt;&gt; )</td>
</tr>
</tbody>
</table>

\[ \text{sel} \]

<table>
<thead>
<tr>
<th>binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[0]}}# )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(X's in don't care positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[XIX]}#} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

\[ \text{cx} \neq \text{cx} \]

\[ \text{cx} \neq \text{cx} \]

\[ \text{d}
\]

\[ \text{cx} \]

\[ \text{name} \]

\[ \text{number} \]

end

begin

1.2. Semantics

The YALLL programmer may use the machine's built-in register names to designate transfer operands, or may bind symbolic names to them, and use the latter in a microprogram. This symbolic binding gives the programmer the opportunity to specify types for his variables. The type indicates

\[ \text{number} ::= \left[ \begin{array}{c}
\text{decimal number} \\
\text{octal number} \\
\text{hexadecimal number}
\end{array} \right] \]

<table>
<thead>
<tr>
<th>relop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; )</td>
</tr>
<tr>
<td>( &lt;= )</td>
</tr>
<tr>
<td>( &gt;= )</td>
</tr>
<tr>
<td>( &lt;&gt; )</td>
</tr>
</tbody>
</table>

\[ \text{sel} \]

<table>
<thead>
<tr>
<th>binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[0]}}# )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(X's in don't care positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[XIX]}#} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

\[ \text{cx} \neq \text{cx} \]

\[ \text{cx} \neq \text{cx} \]

\[ \text{d}
\]

\[ \text{cx} \]

\[ \text{name} \]

\[ \text{number} \]

end

begin

1.2. Semantics

The YALLL programmer may use the machine's built-in register names to designate transfer operands, or may bind symbolic names to them, and use the latter in a microprogram. This symbolic binding gives the programmer the opportunity to specify types for his variables. The type indicates

\[ \text{number} ::= \left[ \begin{array}{c}
\text{decimal number} \\
\text{octal number} \\
\text{hexadecimal number}
\end{array} \right] \]

<table>
<thead>
<tr>
<th>relop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; )</td>
</tr>
<tr>
<td>( &lt;= )</td>
</tr>
<tr>
<td>( &gt;= )</td>
</tr>
<tr>
<td>( &lt;&gt; )</td>
</tr>
</tbody>
</table>

\[ \text{sel} \]

<table>
<thead>
<tr>
<th>binary number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[0]}}# )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(X's in don't care positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[XIX]}#} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ^{\text{[6-0]}%} )</td>
</tr>
</tbody>
</table>

\[ \text{cx} \neq \text{cx} \]

\[ \text{cx} \neq \text{cx} \]

\[ \text{d}
\]

\[ \text{cx} \]

\[ \text{name} \]

\[ \text{number} \]
how many of a register's bits are to be considered significant and how conversions, if any, are to be done. The YALLL/VAX compiler does not support the type checking implied by strict typing, nor all the coercions implied by mixed type arithmetic. It is felt that, at this low a level, the former would be more of an encumbrance than an aid, and that the latter requires too much run-time overhead. In this implementation, the register type determines how much data is transferred on a memory access, and whether a quantity will be sign- or zero-extended or unchanged in the course of a move operation. A limited amount of type checking is done, so that one may add a short type into a longer operand, but not vice versa. Shift operations are also checked, to make sure the source of bits (not the shift count) is no larger than the destination. The only real typing problem occurs in the case of shifts; a byte circular shift, for example, is not what the term implies, but really a byte, extended into a longword, then rotated. The VAX rotation hardware actually only supports sixty-four-bit rotates, so that even a thirty-two-bit arithmetic shift is not as efficient as one would hope. The section on VAX peculiarities gives more detailed information on registers, variable representation, and coercion action.

1.2.2. register transfer operations

Register transfer statements take three forms: memory access, register-register transfers, and three-register arithmetic. Memory accesses specify a source or destination register and a main memory address. This address is a constant, a register content, or the sum or difference of a register content and a constant. The amount of data transferred depends on the declared type of the target register. On the VAX, this defaults to a four-byte longword. A VAX memory reference is normally to virtual memory, since the
addresses passed from the macro-program are usually virtual addresses, and since the memory-mapping mechanism is quite easy for the microprogram to invoke. To avoid this address mapping the keyword physical should appear in the accessing statement.

A register-register transfer can be either a complement or a move. The source of data may be a register or a constant. Move is the only statement for which VAX microcode will be generated to do type conversion.

The three address register instructions provide most of the normal dyadic functions (addition, subtraction, logical and, or, exclusive-or), some shift operations (arithmetic left or right, logical left or right, circular long-word left or right), as well as add1 and sub1. (The latter two compute dest = src1 ± src2 ± 1, for implementing multiple-precision arithmetic.) On the VAX, a negative shift count does a shift of the same type (arithmetic, logical, circular) in the opposite direction to that specified.

By appending encc, for enable condition code, to one of these transfers, the machine's condition code bits may be set. On the VAX, only PSL bits N and Z are affected.

1.2.3. control operations

YALLL provides no code-structuring facilities such as compound statements or looping constructs; all of its control mechanisms are very simple: goto's, subroutine call, return, table jump, and exit. The unconditional goto is the jump statement, and takes a label or constant destination. A conditional jump is of the form "jump label if condition", where the condition is the comparison of a register's contents with those of another register, or
with a constant, or the test of a single bit. The subroutine call-return mechanism is simple and parameterless. The call statement causes the return address to be saved in a sixteen-deep stack, and the rtn causes the top address on this stack to be popped and used.

A more interesting construction is the jtab multi-way branch, where a field of a register is used to select one of several addresses as a jump destination. On the VAX, the width of the selecting field may be up to four bits. The mapping of integer field values to labels is given on the lines between the jtab and etab, the closing bracket. Each line is of the form of a comma separated list of values, a colon, then a label, which is the jump destination if the selected field takes on any of the corresponding values. Besides integers, the value list elements can be selectors, which have the form of binary numbers, but with X's in "don't care" positions. Thus "#XX1" is equivalent to the list "1, 3, 5, 7". The last line of the value-level map may be "else label", which specifies that for any values not specified on the preceding lines, control should transfer to the given label. If the else is not specified, and if the selecting field takes on an unmapped value, execution falls through to the next executable statement.

The exit statement causes execution of the user's microcode to end, and macro-instruction fetching and interpretation to continue. If no argument is given, sequential instruction processing is assumed. If an address argument is given, it is used as a macro-program address, from which the next instruction is taken; the program counter, PC, is also loaded at this time. In VAX microcode, an exit without an argument causes the PC to be incremented by

---

1On the VAX, arithmetic tests take into account the sign bit, so the result of comparisons of long unsigned quantities may be wrong. For example, unsigned FFFFFFFF (hex) > 0, but taking the sign into account gives the opposite result.
one, and the current op-code to be discarded.

1.2.4. other pseudo-ops

In addition to the register name equating statements described earlier, YALLL provides a handful of pseudo-ops to ease symbolic microprogramming. The \texttt{equ} statement serves to equate a name with a constant expression. The expression may include numbers and previously defined constant names; it should not include register names, nor labels (on the VAX, these are not given address values until after all code generation). The \texttt{org} statement allows one to assign an address to the beginning of the code generated by the following YALLL statement. This is often necessary for linking to a machine's native microcode, which generally jumps to a fixed location to enter the user's code.

The \texttt{begin} and \texttt{end} pseudos provide a means for controlling the scope of variable and constant names. These symbols obey the usual block-structure visibility rules under control of \texttt{begin} and \texttt{end}. Labels, however, are global. This means that the same register may be used with different names and types in separate (non-nested) parts of the program. (Recall, though, that a routine call and return from one area to another using the same registers does not cause the saving or restoring of them.) An \texttt{end} statement also denotes the end of the program text, and must be the last statement in it. \texttt{End} causes no code to be generated, and should not be confused with \texttt{exit}.

1.3. VAX Peculiarities

Throughout this language description, I have tried to indicate which features are machine-dependent and, by implication, which are not. The most implementation-dependent features of YALLL/VAX are described in this
section. These peculiarities fall into four groups: representation of short types and conversions; register names; control store addresses; and an assembler escape.

1.3.1. representation

In the VAX, all registers are thirty-two bit longwords. Words and bytes are sixteen and eight bits, respectively. In code produced by YALLL, all register quantities are represented as longwords, for the following reasons: The general, macro-program visible registers have the capability of storing partial register quantities, leaving the upper bits unchanged. However, none of the other registers in the CPU share this ability, so that, in order to avoid propagating garbage when doing operations between register types, one would have to mask or sign-extend quantities coming from the general registers. In order to avoid this overhead, YALLL/VAX always writes full-register results into them. No run-time bounds checking is ever done. So, for byte register x:

```
move x,255
addl x,x,255
```

will cause x to contain 511, even though this quantity cannot be stored in a byte.

Sign "extension" is done, strangely enough, when moving a long quantity into a shorter quantity. Since all variables are represented in longwords, it is not necessary to change representation when moving from short to long. But going the other direction, long quantities are truncated, and sign or zero filled, to assure that the type of the receiving variable is not violated by the transfer. The type conversion for all combinations of source and destination types are shown in Figure 3. (These actions are coded into a table in the
translator, and may easily be changed by recompiling it.)

1.3.2. register names

The register names currently available to the YALLL/VAX programmer are shown in Figure 4, along with the location of their associated physical registers. These names are entered into the symbol table upon initialization of the translator. They all have default types of unsigned long, and are unreserved. ID bus registers are more expensive to access than are other registers, and some are read-only (see Appendix A).

The OPERAND register is the specifier byte of the instruction buffer; this is treated as a pseudo-register. Each time it is read, the byte is cleared, PC incremented, and the next instruction-stream byte shifted into place. The programmer should not, therefore, modify the PC to account for the macro-instruction argument(s).

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ul</td>
</tr>
<tr>
<td>ul</td>
<td>√</td>
</tr>
<tr>
<td>sl</td>
<td>√</td>
</tr>
<tr>
<td>uw</td>
<td>√</td>
</tr>
<tr>
<td>sw</td>
<td>√</td>
</tr>
<tr>
<td>ub</td>
<td>√</td>
</tr>
<tr>
<td>sb</td>
<td>√</td>
</tr>
</tbody>
</table>

where

ul - unsigned longword  √ - no change
sl - signed longword    0 - truncated and zero-filled
uw - unsigned word      ± - truncated and sign-filled
sw - signed word
ub - unsigned byte
sb - signed byte

Figure 3 - Coercion Actions
1.3.3. addresses

The areas of control store designated for user microprogramming are locations 10E0, and 1400-1800 hex. These are the only addresses the translator will attempt to bind to a microinstruction, and are the only addresses which should appear in org statements.

1.3.4. assembler escape

The YALLL language is not designed to allow one to use all the machine's resources, but only to make the writing of microprograms a reasonable task. Therefore, one might want to embed segments of microassembly language code in a YALLL program, either because the code emitted by the translator is unsatisfactory, or because there is no way of dealing with VAX-specific
mechanisms (such as interlock read/write, or the accelerator). To make this possible, one can write VAX microassembly statements between `asm` ... `msa` brackets. The primary restriction on such statements is that one may not use DEC's macro definitions. (One may, however, write one's own macros and use the C compiler's pre-processor to expand them.) The form of an assembly-language statement is:

```
[label:] [const:] field-id/field-value [, field-id/field-value]
```

where the constant binds this word to a specific control store address, and `field-value` is either a compile-time constant, or a label (in the case of the `J` field). Assembly-language statements may be broken over several lines, so long as there is at least one field-value pair, with a trailing comma, on each line.

Finally, each address restricter, for constructing jump tables and subroutine linkages, takes the form `=[01X]` and must be matched by a closing bracket `=end`. For example, to jump to location "A" if the middle sixteen bits of register r3 are zero, else location "B", code:

```
asm
  SPO/43X, ALU/OF, AMX/0, DK/8, SHF/0 ; D_R[R3]
  BEN/18X ; D.BYTES?
  =1001 ; D<23:8> = 0
  J/A ; D<15:8> ≠ 0
  J/B ; D<23:16> ≠ 0
  J/B ; D<23:8> ≠ 0
=end
msa
```

2. VAX-11/780 MICRO-PROGRAMMING SYSTEM

Creating a microprogram for the VAX is a process of several steps, involving various software tools. The primary tools are an editor, the YALLL
compiler, and the console-resident microdebugger. In this section I shall describe the user's interaction with most of these, and detail the choices offered by them.

The excellent editors (ex and vi) available on CS VAX/UNIX are written by Bill Joy [Joy 77b], and should need no introduction to anyone familiar with the system. One of these editors should be used in preparing the YALLL source file. The next step is to have this file compiled.

2.1. The YALLL Translator

The translator is a large 'C' program called yc. It will take an input file, and produce a binary output file. It will not produce a source code listing, but will dump the intermediate code at various points in the processing. Although the translator is actually a single program phase, it conceptually has three passes: the first reads the source, parses it, and generates intermediate code; the second does peephole code improvement; and the third assigns addresses to each microinstruction, and writes the binary file. The translator command line is:

```
yc inputfile [-d[1][2][3]] [-2] [-o filename]
```

where the order of parameters is not significant, except that they are scanned from left to right. The inputfile is the source file, produced by an editing session. It is conventional to use filenames ending in "..m" (as in "source.m") for microprogram sources. If no filename is given, standard input is read until a fatal error or end-of-file (control-d).

The -d options specify that a dump of the intermediate code is to be produced after the specified pass(es) of the translator. This is a human-readable representation of the binary being produced, and is written on the
standard output file. The reading of dumps will be fully explained in a later section. The flag -2 specifies that the second compiler pass, code improvement, is to be suppressed. This should only be done if you feel that the microcode produced by the compiler is incorrect because of "improvements" made in the second pass. Note that microprogram segments entered in assembler-escape mode will not be touched by the code improver in any case.

Finally, the -o parameter governs the binary produced by yc. If this is -o -, no binary file is written. Otherwise, the following word of the command line is taken as a filename, and output is written in it. If no output disposition is specified, a binary file is written in file m.out.

2.2. Macro Processing

The assembly escape provided in the YALLL translator permits the microprogrammer full access to the microarchitecture, including functions not employed by programs written in the YALLL language. However, assembly language programming using this facility is not as easy as programming with DEC's macro assembler. This task may be made easier by the use of the C pre-processor.

The C preprocessor allows one to define one-line macros, with or without parameters. A parameterless macro (such as a constant) is defined by:

```
#define name string
```

And a macro with parameters as:

```
#define name(parameters list) string
```

For example:
To use the preprocessor to do macro expansions in a source file \( z \), then translate the result using yc, the command is:

\[
\text{cc -E } z \text{ } \text{yc options}
\]

One problem with this system is that it does not permit context-dependent constant names, as DEC's assembler does. Thus, defining "$\text{#define RAMX 1}$", so that one may write "$\text{AMX/RAMX}$" will only cause trouble when 1 is substituted for the field name in "$\text{RAMX/0}$", or the like. One should also avoid YALLL keywords and pre-defined registers names.

Rather than writing macro definitions in each microprogram source file, one may collect them in a file (or files), which the preprocessor will read as input upon encountering a line of the form

\[
\text{#include "filename"}
\]

in its input. Note that the quotation marks are mandatory. The preprocessor does not pass the include directive to its standard output, but does insert several lines of its own. These are ignored by yc, but may throw off line numbers reported in error messages. The C preprocessor will read multiple files, providing an alternate method of including a macro collection. To read and macro process files \( x \) and \( y \), then feed them to yc, the command line is:

\[
\text{cc -E } x \text{ } y \text{ } | \text{yc options}
\]
2.3. Symbol Table and Code Dumps

At several points in the translation process, yc can be persuaded to dump some of its internal tables in human readable form. These are primarily intended for maintenance of the translator, but may also be useful to the programmer, as will be outlined.

A symbol table dump may be obtained at any point during the scan of the source program by the inclusion of a comment beginning ;%. This dump may appear before any semantic action for that line has taken place. The dump has three parts: the histogram, local symbols, and global labels. The histogram gives an indication of hash table usage, and indicates, for each of the 256 table entries which is not empty, how many symbol names hashed to that entry. This statistic includes pre-defined symbols, such as register and field names, which do not appear elsewhere in this dump.

The local symbols are those visible at the point of the dump, according to the normal begin . . . end nesting rules. Global labels are program labels which were defined or first used within a block not including the point of the dump. The dump of each symbol entry gives: the number of characters in the symbol's name; the name; the line number where it was defined (if defined) or first used; the pseudo-line number on which it was first defined or used; its type (error or undefined; register; constant; label; and field name, which are for pre-defined symbols only, and should not appear in a dump); and its value, if defined. For labels, the value is a relative address in brackets. For registers, the value is an index into the compiler's register table. The pseudo-line number is the value kept in a variable set by the PS number pseudo-op. If a high-level language translator were to emit YALLLL, a PS pseudo would mark the beginning of the code for each HLL statement, so
that the source of errors in the YALLL program would be traceable to a HLL statement.

At the end of the dump, one of the messages "open action pending" or "close action pending" may appear. The former indicates that the symbol begin has been scanned but not processed; the latter that end has been scanned but not processed. These should only appear when one writes, for example "end ;%", and may change the meaning of the dump. A symbol table dump may also occur spontaneously, as a result of certain kinds of internal translator errors. An error message will accompany such a dump.

A code dump may be obtained at the completion of any of the translator's three passes as indicated by the -d command line argument. The dump after the third pass gives the most information, as it shows the result of pass two code improvement, as well as address assignment. However, the relationship of YALLL source to assembly-like dump is hardest to see at this point. This relationship is much better shown by the pass one dump, which might be used as an aid to understanding the final code.

The dump exhibits several of the fields of each of the data structures representing a word of generated code. Each micro-node, as these are called, is represented by four attributes: address, field values, branch information, and uses-sets information. The address is given as a "relative" address, in brackets, optionally followed by an absolute address (in parentheses), or constrained address, preceded by an equals sign (=). In pass one and two dumps, absolute locations are the result of org statements, and constrained locations of conditional or table branches, or subroutine calls. By the end of pass three, all locations should be given absolute addresses. Relative addresses are decimal, absolute are hexadecimal, and
constraints binary. It is impossible for a location to have both constrained
and absolute addresses.

The body of a microinstruction, its field values, is given in approximately
the format accepted by the assembler: fieldname-slash-hex-value. Note, however,
that the value given for the CID field is only that of the four bits not
overlapping fields ADS nor FS. Fields not explicitly set during code genera-
tion, and thus taking their default values, are not shown. As an example, the
pass one dump of the code generated by \texttt{x: jump \textit{x} if r0 = t0} is shown in Fig-
ure 5.

**CODE DUMP**

code size = 5 words

[0]: SPO/20, QK/8, ALU/e, BMX/4,
\(J/\{1\}\)
USES: LatchC RegC
SETS: Qreg LatchC

[1]: CCK/1, SPO/40, ALU/8, RMX/0, AMX/0, BMX/7,
\(J/\{2\}\)
USES: Qreg LatchAB RegAB
SETS: LatchAB CondCode

[2]: BEN/1b,
\(J/\{3\}\), 2-way branch
USES: CondCode
SETS:

[3]:=1011:
\(J/\{5\}\)
USES:
SETS:

[4]:=1111:
\(J/\{0\}\)
USES:
SETS:

Figure 5 - Code Dump
The jump address of each microinstruction is determined by the settings of the SUB, BEN, and J fields. The former two are shown with the other microword fields. The J field is given last, and indicates either a [relative] or (absolute) addresses. By the end of pass three, all jumps, like all addresses, should be absolute; if a relative jump appears in a third pass dump, a label reference did not get resolved, and a run-time error will occur. For a multi-way branch (BEN not zero), an indication is given of the number of possible destination addresses for the jump.

For the purpose of pass two code improvement, the generation routines store in the micro-nodes not only the appropriate field values, but also the names of the resources (registers and latches) which are being manipulated by the microinstruction under construction. This appears on the two lines of uses-sets information of the dump of the micro-node. A resource is used if its value is used in a calculation, and set if a new value is clocked into it. The words EXCLUSIVE_USE appear when a microinstruction should not be combined with those around it. This is the case when the assembly escape has been used, or when the generated code is sufficiently tricky that any attempt at code improvement might change its meaning. A micro-node represented by only the relative address and an "X" is one which has been deemed unnecessary by the code improver. These may be considered to have been deleted, as they are not assigned addresses, nor written on the output file.

2.4. Other VAX Microcoding Utilities

The binary file produced by yc is not ready to be loaded into WCS, but must first be linked with other routines to be loaded there. The output of yc contains no explicit address information; the first twelve bytes belong at control store location 10E0, and the following twelve-byte words are to be loaded
beginning at 1400 hex. The LSI-11-resident WCS loader can only load into sequential locations, and (because of the floppy disk file format), only in multiples of 512 bytes. Also, the console resident debugger requires a floppy disk resident image of WCS (see section 2.6). Thus it is desirable to make a single file containing the entire WCS image: both native and user-written microcode. The native microcode can be found in a floppy disk file, currently WCS118.PAT. This may be copied to a Unix file using arff, the floppy file utility. The program merger may be used to combine a Unix file copy of the native microcode (the "system file") with yc output ("user file") into a combined file ("target file"). The merger program will prompt the user for the appropriate file names.

To verify the performance of yc and merger, two versions of a dump utility are available, to interpret binary microcode files. The program undo will dump a named file (default m.out) in microassembler-like format. This file is assumed to be the output of yc. The dump format is the same as that of a translator code dump, except that: all fields are shown, as the field use information has been lost; resource uses-sets information has, similarly, been lost; words are given in increasing order of address, which may have little to do with logical order. Similarly, interdump will verify a merger operation by dumping, in the same format as undo, requested addresses of the file named in the command line. Since the merged WCS image contains at least a thousand words, dumping the whole file would be impractical, thus it is done interactively.

Under the system described here, all WCS files are loaded by the console LOAD/WCS command. Thus, these files must be written on the floppy disk, in the format understood by the LSI-11 operation system. Two VAX programs
make this possible: The floppy disk device driver is part of the Unix kernel, and deals with transferring sectors (of 128 bytes) to and from the floppy. However, this program knows nothing of the disk format other than the sector size, and treats it as one long, sequential file. Thus it should not be used alone; a command such as "cat /dev/floppy" is almost certainly wrong.

The program arff, written by Keith Sklouer, deals properly with the disk format and uses the Unix floppy driver to request the data transfers of the LSI-11 program (which actually deals with the device). Arff is meant to appear to the user like the program tar, and is invoked as

    arff actioncode [filename ...]

where actioncode is one of the following:

t - list which of the named files are listed in the floppy directory. If no filename arguments are given, the name of each file on the floppy is printed.

tv - like t, but more information is given with each listing, such as creation date, and size of the file, in blocks. Also, the number of directory entries remaining is printed, and, if listing the whole directory, the size of unused areas.

x - extract named file from floppy to Unix file. If the file name is a path name (with slashes), the last portion is taken to be the floppy file, and the entire qualified name is the Unix file.

r - replace (or add) floppy file from named Unix file. Qualified file names are interpreted as for x. If the named Unix file is larger than an existing floppy file of the same name, it may be necessary to first delete the floppy file, forcing an add action, rather than a replace. If the directory is full, or the whole floppy disk is full, an error message will be written.
d – delete named files from floppy directory.

For example, to repack the floppy, combining several small unused areas into a larger unused area, we could extract the entire contents, delete it all, then replace each file as follows:

\[
\begin{align*}
    \% \text{ set } \textit{fl} &= \textquote{arff t} \quad \text{C-shell variable } \$\textit{fl} \\
    \% \text{ arff x } \$\textit{fl} \\
    \% \text{ arff d } \$\textit{fl} \\
    \% \text{ arff r } \$\textit{fl}
\end{align*}
\]

Floppy disk file names have up to six characters, optionally followed by an extension (qualifier) of up to three characters; e.g. WCS118.PAT, WCSMON.HLP. Furthermore, they must be composed only from the Radix-50 character set: A–Z, 0–9, $, %, and period (.). The latter character should be avoided, since it is also the separator between name and extension. Arff does case translation, so that all letters appear to be in lower case. WCS image files should have names of the form WCSnnn.PAT, since there seems to be some restriction on the names of these files, and this formula seems to work.

2.5. Dealing With UNIX

In order to load and debug microprograms on the VAX, it is necessary to stop timesharing, halt the machine, and either run stand-alone, or use Unix as a single user. Here we will outline some of the important Unix commands necessary for failure-free operation. It is assumed that you have arranged with the system manager to halt timesharing, and that you have a Unix account, though not necessarily root.

To increase disk through-put, Unix employs file read-ahead and write-behind. Because of this, if the system goes down unexpectedly, grave file-system inconsistencies may result. The sync command causes all disk files to be brought up-to-date, by writing out buffers-full of data destined for
writing. This command should be used whenever the system might stop, as when you are about to halt the machine, or test new or shaky microcode under Unix. See the *Unix Programmer's Manual*, sync (1 & 2), and update (8) for details. Unix is brought up from a halted machine using the console command B or @UNIX in response to the >>> prompt. This causes console commands to be executed from the floppy disk file UNIX. Finally, a VAX boot program is read from the hard disk, and will prompt file:, after which type unix, or whatever the appropriate name is for the system being run. This will be loaded, report on the available memory, and prompt with #. If a control-d is typed to this prompt, commands will be taken from the file /etc/rc, which will bring up timesharing. Therefore, don't type gratuitous control-d's!

Before bringing up timesharing, you should always check the integrity of the file system. This is done with the chk command (which in turn executes dcheck and icheck, see section one of the *Unix Programmer's Manual*): chk /dev/rrp0a /dev/rrp0g is the least you should do. This takes about fifteen minutes, and will report on any inconsistencies it finds. If dcheck reports a file having more entries than links, the system manager should be notified to fix this situation before you go any further. If you did sync's before taking the system down, there should be no problems.

If you don't wish to bring up the full system, but want to access files in your directory, it will be necessary to mount the /usr file structure with

```
# /etc/mount /dev/rp0g /usr
```

The easiest thing to do at this point is to use the `login name` command to give you your own home directory, shell, and identity. An alternate possibility, which retains the root user identity, is the following:
The problem with this is that any files you create in your directory belong not to you, but to root.

### 2.6. At the Console

Loading and debugging microprograms from the VAX consol requires using DEC-supplied, LSI-11-resident console software. These programs are quite adequate for most of the simple tasks necessary for debugging a microprogram.

The console command interpreter takes the place of front panel lights and switches on the VAX. When the console is in "program I/O mode" (communicating with a VAX program, as when waiting for a login), it may be switched to console command mode by typing a control-p. The console prompt is `>>>`, character erase is `del` or `rubout`, line kill is control-u, program kill is control-c. The command `SET TERMINAL PROGRAM` is the inverse of the control-p command — it returns the console to program I/O mode. A `HALT` command stops VAX CPU instruction interpretation, and puts the microcode in a console-command servicing loop. The microcode's cooperation is necessary, for example, to manipulate main store locations from the console. The `CONTINUE` command restarts VAX instruction execution where it left off, and `START address` first sets PC to `address`, then starts instruction execution. Many other commands are available, and are outlined in the console floppy file `CONSOL.HLP`. This may be typed at the console by the command `@CONSOL.HLP`, or transferred to a Unix file with arff and listed; see also Figure 6a.
VIRTUAL EXAMINES AND DEPOTS

ADDRESS RELOCATION REGISTER IS ADDED.

To effect the address of a Phys Mem.

- RELOCATION REGISTER ADD.

- SET CUP PROG ADDRESS.

- PHYS MEM.

- PHYS 0.

- SET PHYS MEM ADDRESS.

- SET PHYS MEM ADDRESS.

- ACTIVE ADDRESS.

- ACTIVE ADDRESS.

- SET CUP ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.

- CLEAR ADDRESS.
disk-resident image of control store which it can inspect; it is used to
in, but not read from. Thus, the WCS microdebugger requires that there be a
processor, writable control store is a write-only medium — it can be written
the WCS MON. HLP is shown in Figure 6b. From the point of view of the console
debugger is called by WCS, and programs WCS. It is command summary, from
where the START option indicates the lowest address of the load. The

```
>> LOAD/WCS/START:1000
```

phosph by the command
load a WCS image and invoke the microdebugger. Loading is acco-

The commands directly relating to microprogramming are those that

Figure 6a - Console Command Summary

```
E) CUTION WILL TERMINATE
H) ALT TYPE OR OPERATOR TIPS A (COMMAND FILE EX-
EXTENDED VERSION OF THIS COMMAND FOR THE VAX-11/780
EXTENSION WILL CONTINUE TO OPERATE AS IF IT WERE A VAX-11/780 COMMAND. PROGRAM CONTENTS ARE NOT MAPPED INTO THE
UNIT A A NON-SIGAL IS RECEIVED FROM THE
COMMAND WITH LARGE COMMAND FILE EXECUTION TO STOP
WHEN EXECUTED FROM AN INTEGRATED COMMAND FILE.
-CAUSES A CONSOL SOFTWARE REBOOT.
-ENABLES A USER TO ENTER DEBUGGER HELP.
-TURNS MICRO-DEBUGGING FOR DEBUGGER HELP.
-REPEAL ANY-CONSOL COMMAND. - CAUSES THE CONSOL TO REBOOT/EXECUTE
-EXECUTE A FILE OF INTEGRATED COMMANDS PREVIOUSLY
-CONTROL REPLACEMENT (SEE PERFORM)
-EXECUTE A FILE OR A SPECIFIED COMMAND.
-PRINTS THE CONTENTS OF THE CURRENT LOCAL PROGRAM
-PRINTS THE CONTENTS OF THE LOCAL PROGRAM
-SHOWS THE LOCAL PROGRAM
-PURE PROGRAM CONTENTS.
-PRINTS THE PROGRAM CONTENTS.
-SET TERMINAL PREFIX TO # OF BLANKS UNIVERSITY
-GENERAL PURPOSE DUMP TO TERMINAL
-TURNS OFF PROGRAM
-TERMINAL COMMANDS
-SET DEFAULTS.
```

The commands directly relating to microprogramming are those that

specify the starting address of a load (an ordinate also useful to

NOTE: THE START ADDRESS MAY ALSO BE USED TO

LOAD/WCS/FILENAME.

LOAD FILE TO MAIN MEMORY.

LOAD FILE TO INTEGRATED COMMAND FILE.

PRINTS THE CURRENT LOCAL PROGRAM.

PRINTS THE CURRENT LOCAL PROGRAM.

PRINTS THE PROGRAM CONTENTS.

PRINTS THE CONTENTS OF THE CURRENT LOCAL PROGRAM.

PRINTS THE CONTENTS OF THE LOCAL PROGRAM.

SHOWS THE LOCAL PROGRAM.

GENERAL PURPOSE DUMP TO TERMINAL

TURN OFF PROGRAM

TERMINAL COMMANDS.

SET DEFAULTS.
MICRO-DEBUGGER HELP FILE
REV-0 MAY 1977

TO STOP PRINTING, TYPE C

DEBUGGER COMMANDS (ALL TERMINATED BY CARRIAGE RETURN)

'E/P<ADDRESS>' - EXAMINE PHYSICAL MEMORY

'E/ID<ADDRESS>' - EXAMINE ID BUS REGISTER

'E <ADDRESS>' - EXAMINE WCS LOCATION, DISPLAY ALL FIELDS

'E<ADDRESS><FIELDNAME-1>.<FIELDNAME-2>....<FIELDNAME-N>
  EXAMINE WCS LOCATION, DISPLAY ONLY FIELDS
  THE FIELDS SPECIFIED

'CONnNUE' - RESUME MICRO-INSTRUCTION EXECUTION AS SPECIFIED BY CONTENTS OF MICRO-PC(UC)

'START<ADDRESS>' - START MICRO-SEQUENCER AT <ADDRESS>

'HALT THE MICRO-SEQUENCER

MICRO-SEQUENCER

INSTRUCTION EXECUTION THEN HALT THE MICRO-SEQUENCER WITH ACTIVE ONE MICRO-INSTRUCTION EXECUTION STEP MODE.

-HALT THE MICRO-SEQUENCER

-HALT THE MICRO-SEQUENCER

NOTE: DEPOTS TO THE REGISTERS(A) ARE NOT SUPPORTED.

-CONTINUE MICRO-SEQUENCER

DEFINITION OF REGISTERS

The following REGISTERS are supported:

-RC<ADDRESS> DATA TO AN RC REGISTER

-RC<ADDRESS> DATA TO AN RC REGISTER

-RC<ADDRESS> DATA TO AN RC REGISTER

-D/RA<ADDRESS> <DATA> TO AN R REGISTER

-D/RC<ADDRESS> <DATA> TO AN R REGISTER

-D/SYMBOLIC-NAME<ADDRESS> <DATA> TO ONE OF THE SYMBOLICALLY NAMED REGISTERS

-RETURN' - RETURN TO THE CONSOLE PROGRAM

-OPEN <FILENAME>' - OPEN SPECIFIED FILE ON FLOPPY DRIVE 0

-OPEN <FILENAME>' - OPEN SPECIFIED FILE ON FLOPPY DRIVE 1

-OPEN SPECIFIED FILE ON FLOPPY DRIVE 1

-OPEN SPECIFIED FILE ON FLOPPY DRIVE 1

NOTE: 'OPEN' IS USED TO SPECIFY A FILE CONTAINING THE MICRO-CODE CURRENTLY LOADED IN THE WCS PORTION OF THE CONTROL STORE.
Figure 6b - WCS Debugger Command Summary

examine and patch microcode. (This is another reason for building a micro-
code image on the floppy, rather than going the shorter route of loading con-
trol store directly from the VAX.) The WCS command OPEN filename permits
the debugger to use the named file, which it assumes has already been
loaded, as shown previously. Modifying control store using the D command
changes both the control store contents and the disk file. The format for
examining and changing microstore locations is similar to the dump and
assembly formats, except that field name is separated from value by a space
for setting values and an equals sign for examining them (rather than a
slash); CID overlaps FS and is only four bits, and MCT overlaps FS but not
ADS.

A HALT to WCS will stop microinstruction execution, and is necessary
before depositing in control store or registers.

A very useful feature for debugging is the ability to set a break point in
the microprogram. The break point may be set either in console command
mode or when running the microdebugger, by placing the break address in
the micro-break ID register (21), then enabling the facility by the command
SET SOMM. When the given location has been executed, the microprogram
will halt, and a message will be printed. When you are done using this facil-
ity, it must be disabled by the command CLEAR SOMM. See the tutorial
example for a use of the break point.
2.7. A Tutorial Example

This section gives a step-by-step example of the microprogram entry process, including use of the translator, loader, and console microdebugger. The example program will be a simple emulator, shown in Figure 7a. (This is very similar to the CS-152a and CS-292R SM-1 example for the HP 21-MXE computer.)

The simple computer emulated has an accumulator (AC), MAR, MBR, and PC, and 1024 words of (16-bit) memory. RAB register r0 will be used for AC, r1 for MAR, r2 for MBR, r3 for PC, and r4 will point to the area of memory to be used as the emulated memory. After these registers are loaded with the appropriate initial values, the microprogram is called by the XFC instruction; in assembler this is coded as "\texttt{.byte \textendash0xfe}". When the program is done, it will return its values in the same registers and in memory. The emulation will use word addressing, and will interpret eight instructions. This source program should be typed into a file, call it sm1.m, using a Unix text editor.

2.7.1. translating the program Once the source is correctly entered, it should be translated using \texttt{yc}. To get an idea of the code produced, try

\begin{verbatim}
yc sm1.m -d13 | lpr
\end{verbatim}

The pass one dump should be pretty easy to partition into the groups of instructions produced by each source statement. The pass three dump is much more convoluted. This, though, is the appropriate listing for debugging at the console.

Once the program has been translated, it must be combined with the native microcode to produce a new WCS image file, to be written on the floppy disk. If a copy of the native code is not available as a Unix file, one
Emulator for the SM1

This program emulates a computer having eight instructions plus halt. It uses VAX general registers r0 - r3 as the emulated AC, MAR, MBR, and PC, and register r4 to point to the 1k word emulated memory in VAX memory. The XFC instruction takes no parameters except in registers; initial values for AC, PC, and the memory space pointer.

org 10E0
reg AC = r0 signed word
reg MAR = r1 unsigned word
reg MBR = r2 signed word
reg PC = r3 unsigned word
reg base = r4 unsigned long
reg temp = Qreg

I fetch:
all temp,PC,1 : byte-to-word address factor
add VA, temp, base
load MBR,VA ; MBR <- mem[base+PC]
: defer incrementing PC till end of execution cycle
and MAR, MBR, 3FF ; extract address from instruction
: now do case jump on op-code
jtab MBR<15:12> of
0:Cmpl
1:Shr
2:Bneg
3:Jmp
4:Store
5:Load
6:And
7:Add
else Halt
stab

Halt: add PC,PC,1
exit

C mpl:
cmpl AC,AC ; complement instruction
jump inc_pc

Shr:
srl AC,AC,1
jump inc_pc

Bneg:
: branch if (AC)<0
jump inc_pc if AC >=0

Jmp:
: unconditional branch
move PC,MAR
jump I fetch
Store:

; store instruction
all temp, MAR, 1; word-to-byte address factor
add VA, temp, base
stor AC, VA
jump Inc_pc

Load:

; Load instruction
call Fetch_mem
move AC, MBR
jump Inc_pc

And:

; Logical AND instruction
call Fetch_mem
and AC, AC, MBR
jump Inc_pc

Add:

; Addition instruction
call Fetch_mem
add AC, AC, MBR

Inc_pc:

add PC, PC, 1
jump Fetch; its only safe to increment PC after all fetches
; for this instruction are over, so a memory fault (page absent)
; won't cause us to restart in the wrong place.

Fetch_mem:

; data fetch routine
all temp, MAR, 1
add VA, temp, base
load MBR, VA
rtn

end

Figure 7a - SM-1 emulator example, YALLL source

should be obtained by running

% arff x wcs118.pat

Next, run the merger program:

% merger
System file name: wcs118.pat
User file name: m.out
Target File name: wcs101.pat

Check some locations of the resultant file with interdump wcs101.pat.
Compare them with the pass three dump. Those fields shown in the dump should also have the same values in the disassembly. Now, write the WCS file on the floppy disk with

```
% arff r wcs101.pat
```

If more than one microprogram image is to reside on the floppy at a time, they obviously must have different names. So you may need to change the name in these examples, if more than one microprogram is being prepared.

### 2.7.2. running stand-alone

Now, we will bring the system down, and test the microcode stand-alone. Even in this mode of operation there are several things which have to be taken care of. The most pressing of these is the system control block (SCB). In the procedure that follows, this is set up to halt the processor on any error. The halt address will be the SCB vector address, which is determined by the source of error (see Figure 5 of Appendix 1).

On the console, login to Unix. Make sure that everyone else has logged out (using the `who` command). Insure the integrity of the file system by issuing the `sync` command two or three times. Halt the VAX by typing control-p, to which the console will respond with the `>>>` prompt. Answer this with H return. The VAX should now be halted. Load your WCS file with the command `LOAD/WCS/START:1000 WCS101.PAT`. This should take a few seconds and respond that 8000 bytes were loaded.

In order to set up the VAX for stand-alone program operation, type the `INIT` console command. Among other things, this turns off memory mapping, so you do not have to worry about page tables. Now, type the following commands:
Set up a small VAX program, and a small SM-1 program:

```plaintext
>>> D 100 FC  
VAX program - XFC; halt
>>> SET DEF WORD
>> D 200 5003  
SM-1 program - load 3
>>> D + 0      
complement
>>> D + A000   
halt
>>> D + 00FF   
location 3: data
>>> SET DEF LONG
>>> D/G R3 0   
SM-1 PC
>>> D/G R4 200 
SM-1 memory base
>>> D/G R0 BBBB 
SM-1 AC - trash
```

Now run the interpreter program:

```plaintext
>>> START 100
```

This should quickly halt, with PC = 102. Examine some registers:

```plaintext
>>> E/G R0  AC, should show FFFFFFF00
>>> E/G R4  base address, should still be 200
>>> E/G R3  PC, should show 3
```

Now that this worked (or even if it didn't), set a break point at the case instruction decoding the op-code, and watch it perform the complement operation.

```plaintext
>>> D/G R3 1  
SM-1 PC to complement
>>> D/ID 21 1421  
micro-break at complement
>>> SET SOMM  
enable breakpoint
>>> START 100
```
The machine should halt with micro-PC = 1421. Enter the WCS debugger:

```wcs
>>> WCS
WCS> OPEN WCS101.PAT
```

Look at the D register, which should contain the complement opcode, zero. Also, r0, the pseudo-AC, which should still contain FFFFFFF00.

```wcs
WCS> E DR
WCS> E RA 0
```

Single step through the complement operation. Set single step mode, execute a microinstruction.

```wcs
WCS> SET STEP
WCS> START 1400
WCS> E LA
WCS> E Q
WCS> E RA 0
```

Notice that the register contents, latched into LA, were complemented on the way to register Q. Execute another microinstruction, then look at r0 again.

```wcs
WCS> CONTINUE
WCS> E RA 0
```

Now, look at the microinstruction which did the complement. Modify it so that, rather than a complement, it will exclusive-or register r0 with the constant 1, toggling the low-order bit.

```wcs
WCS> E 1400
WCS> D 1400 ALU 8, BMX 6, KMX 1
```

Recall that this will change the disk file, as well as the control-store location. To single-step through this sequence of instructions again, use the command START 1400, then CONTINUE, examining the registers as before. Finally, change the program back to doing complements, then leave the debugger, clearing the machine.
Invent a program to test all the opcodes.

2.7.3. running with Unix

If your microcode seems to work well, you are ready to try it out under Unix. When running microcode with the operating system, it is possible to write supporting routines in C and assembly language. This is generally easier (and less error-prone) than typing in hex machine codes by hand.

In console mode type:

```bash
>>> @UNIX
file:unix
```

The system should come up, give you a message about available memory, and a # prompt. Type the sync command a couple of times, and halt the VAX as before. Recall that your WCS file, with your microcode, is still in control store. Enable the XFC instruction by changing longword 14 of the SCB (which Unix keeps at physical location zero). Then, bring Unix back up, and log in.

```bash
# sync
# sync
# control-p
>>> H
>>> D/P 14 2
>>> CONT followed by TWO returns
#/etc/mount /dev/rp0g /usr
# login yourname
```

After logging in, you should be able to run C programs with embedded XFC instructions. There are two ways to create such programs. If the microprogram can find its own parameters in the C program, or is parameterless, the XFC may be entered in the C source as
If, however, the microprogram requires parameters in registers, it is usually easier to produce VAX assembly language from the C program, using cc -S, and edit it. For example, compiling the routine of Figure 8a will produce the assembly program of Figure 8b, which can be edited as in Figure 8c, to pass parameters to and from the sm1 emulator program. Try running a more complex example.

After you have run some trials, you should halt the VAX, load the normal microcode, and bring up Unix multi-user. Restarting Unix loads the usual SCB on top of the one you changed so that the XFC instruction will no longer execute your microcode:

```bash
% sync
% sync
% control- p
>>> H
>>> LOAD/WCS/START:1000 WCS118.PAT
>>> @UNIX
file:unix
# chk /dev/rrp0a /dev/rrp0g
# control- d
```

```c
smlroutine( accu , locc, codespace)
int *accu, *locc, *codespace;
{
    register int ac = *accu,
    pc = *locc,
    *base = codespace;

    *locc = pc;
    return( ac);
}
```

Figure 8a - Microcode support C program
LL0:
  .data
  .text
  .align 1
  .globl _sm1rout
  _sm1rout:
  .word .R1
  jbr L13
L14:
  movl *4(ap),r11
  movl *8(ap),r10
  movl 12(ap),r9
  movl r10,*8(ap)
  movl r11,r0
  ret
  ret
  .set .R1,0xe00
L13:
  jbr L14
  .data

Figure 8b - Assembly language produced from Figure 8a

LLO:
  .data
  .text
  .align 1
  .globl _sm1rout
  _sm1rout:
  .word .R1
  jbr L13
L14:
  movl *4(ap),r0
  movl *8(ap),r3
  movl 12(ap),r4
  .byte 0xfc
  movl r3,*8(ap)
  ret
  ret
  .set .R1,0xe00
L13:
  jbr L14
  .data

Figure 8c - Figure 8b edited to pass parameters to emulator
CHAPTER 4

EXAMPLES

In order to illustrate the use of YALLL, some examples are presented here. Each is shown in several forms: YALLL source and object (as shown by a pass three dump) are given for each. In three cases, the latter is turned into DEC macro-code, for comparison with a hand-coded version of the program. Comparisons of various sorts are made with microcode for other machines. In two cases, comparisons are made with the code generated by the YALLL translator for the HP 300, a stack machine with a much simpler, vertical microinstruction. Comparisons are also made with microcode for the HP-21MXE, also a short microword, vertical machine.

1. String Translation

The first example is an instruction to transliterate a string according to a table. This is similar to the IBM-370's TR instruction. The character string is addressed by register 'str', and ends with a null (0) byte. A table is addresses by register 'tbl'. Each byte of the string is examined, and, if not zero, is replaced in memory by the byte in the table which it addresses; that is, memory( (tbl) + char). When a zero byte is encountered in the source string, the microprogram exits. To time this routine, the string "The quick brown fox jumps over the lazy dogs." was translated into upper case. The YALLL source is shown in Figure 9a, and the generated code in Figure 9b. This is expressed in macro-code in Figure 10a, and may be compared with Figure 10b, a hand-coded version. The VAX assembly code this replaces is shown in Figure 11a, and is (almost) equivalent to the C language fragment.
org 10E0
reg str = r0
reg tbl = r1
reg char = t0 unsigned byte
reg mar = VA

loop: load char, str ; get addressed character
      jump out if char = 0 ; test for zero, if zero, go quit
      add mar, char, tbl ; add to table base address
      load char, mar ; fetch character from table
      stor char, str ; replace character in string
      add str, str, 1 ; bump string address
      jump loop ; go do it again
out: exit
end

Figure 9a - String translation YALLL source
Figure 9b - VAX microcode generated from Figure 9a
Figure 10a - Macro-Code of Figure 9b

Figure 10b - Hand-Coded program for Figure 9a

register unsigned char *tbl, *str, c;
while(c = *str) *str++ = tbl[c];

The VAX also has a string translation instruction which, although it won't translate a string in place, is otherwise very similar, as is shown in Figure 11b.

The size and speed comparisons for this example are shown in Figure 13. All timings were made by executing the program segment 100,000 times.
loop:
    movzbl (str).r0
    beq out
    movb (tbl)[r0],(str)+
    brb loop

case jump

out:

Figure 11a - VAX assembler program to translate string

    movtuc $s_length,(src),$0,(tbl),$d_length,(dest)

Figure 11b - VAX instruction to translate string

<table>
<thead>
<tr>
<th>Translate</th>
<th>YALLL HP</th>
<th>VAX HP</th>
<th>Hand-generated Microcode HP VAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>146</td>
<td>128</td>
<td>134 73</td>
</tr>
<tr>
<td></td>
<td>1350</td>
<td>93, 174</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>32</td>
<td>180</td>
<td>28 108</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>11, 11</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12 - Comparison of string translate routines

Then, the user-microcode overhead was subtracted from the time for each microcode example. This is the time it takes to enter user microcode upon the recognition of the appropriate macro-opcode, on the the VAX-11/780 this is about 3.5 microseconds.

2. Pascal Assist Instructions

   The second example (Figures 5 – 8) is a case-jump instruction for use by the Pascal interpreter, PX [Joy 77a]. This instruction grabs the next Pascal opcode byte from the location addressed by register 'lc' (which is incremented after use), and uses it to index into a table of offsets, whose zero-th

* First figure for single instruction, second for four-instruction loop
element is addressed by register ‘tbl’. The contents of this register are added to the two-byte offset fetched to form the target program address. This works very much like the VAX's CASE instruction, but the dispatch table need not follow the instruction, no range checking is done, and the case selector is implicitly the unsigned byte from the PX instruction stream. This instruction is executed at the end of the interpretation of each Pascal instruction, and thus begins the interpretation of the next.

A further PX-assist microprogram is shown in Figures 17 - 20. This combines the above dispatch function with computation of data addresses (L-values) using a lex-level and displacement from the PX instruction stream and a display in memory. Upon interpreter initialization, the display base address is passed to the microcode, which places it in SSP. Under Unix, this register (supervisor stack pointer) is an unused, per-process register. Thereafter, the microcode can be invoked to compute an address using this

```
org 10E0
reg lc = r10 ; PX location counter
reg tbl = r8 ; address of address table
reg opcode = $t1 unsigned byte ; PX opcode
reg t_entry = $Qreg signed word ; offset, from table
reg jaddress = t2
reg lookup = Dreg
reg MAR = VA

dispatch:
load opcode, lc ; fetch PX opcode
sll lookup,opcode, 1 ; shift op to address words
add MAR, lookup, tbl ; add table base - MAR now
                        ; points to table entry
load t_entry, MAR ; fetch displacement from table
add jaddress, t_entry, tbl ; add to table base
add lc, lc, 1 ; update PX location pointer
exit jaddress ; split
```

Figure 13a - PX-assist case-jump YALLL source
10E0 - VAX microcode generated from Figure 13a

```
address microinstruction
(10E0):  VAK/1, SPO/4a, ALU/f, AMX/0, J/(1400)
(1400):  FS/0, CID/8, ADS/0, DT/2, J/(1401)
(1401):  SPO/31, KMX/12, ALU/d, RMX/0, AMX/1, BMX/8, J/(1402)
(1402):  SPO/21, SI/2, ALU/e, BMX/4, SHF/1, DK/8, J/(1403)
(1403):  SPO/48, ALU/5, RMX/0, AMX/1, BMX/3, J/(1404)
(1404):  VAK/1, ALU/5, RMX/0, AMX/1, BMX/3, J/(1405)
(1405):  FS/0, CID/8, ADS/0, DT/1, J/(1406)
(1406):  QK/8, ALU/f, RMX/0, DT/1, AMX/2, J/(1407)
(1407):  SPO/48, ALU/5, RMX/0, AMX/1, BMX/3, J/(1408)
(1408):  SPO/32, ALU/5, RMX/1, AMX/1, BMX/3, J/(1409)
(1409):  SPO/4a, SI/2, KMX/1, ALU/5, AMX/0, BMX/8, J/(140a)
(140a):  SPO/5a, SI/2, KMX/1, ALU/5, DT/0, AMX/0, BMX/6, SHF/0, J/(140b)
(140b):  VAK/1, IEK/1, PCK/1, SPO/22, FS/0, CID/1, ADS/0, ALU/e, BMX/4, BIC/2, J/(ab)
```

Figure 14a - Macro-code for Figure 13b

```
10E0:  VA_R[R10]
      D[BYTE]_CACHE
      RC[T1]_D.AND.K[FF]
      D_ALU.LEFT, ALU_RC[T1]
      ALU_D+R[R8]
      VA_D+LB
      D[WORD]_CACHE
      Q_ALU, ALU_D.SXT[WORD]
      ALU_Q+R[R8]
      RC[T2]_Q+LB
      ALU_R[R10]+K[.1]
      R[R10]_LA+K[.1]
      PC&VA_RC[T2], FLUSH.IB, J/IB.FILL
```

Figure 14a - Macro-code for Figure 13b
DISPATCH:

10E0:

```
VA_R[R10]
D[BYTE]_CACHE
ALU_D.OXT[BYTE], Q_ALU.LEFT,
LAB_R[R8]
VA_LA+Q
D[WORD]_CACHE, ALU_R[R10]+K[.1]
R[R10].LA+K[.1], DT/LONG
LAB_R[R8]
ALU_D.SXT[WORD]+LB,
PC&VA_ALU, FLUSH.IB, J/IB.FILL
```

Figure 14b - Hand-coded Pascal-assist case-jump

```pascal
case (lc)+,$0,$255
    (case table here)

jmp (loop) : register 'loop' points to case instruction
```

Figure 15 - VAX assembly code replaced by microcode of Figure 13

<table>
<thead>
<tr>
<th>Case</th>
<th>YALLL</th>
<th>Hand-generated Microcode</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (msec)</td>
<td>3.7</td>
<td>3.9</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>2.3</td>
<td>2.9</td>
<td>?</td>
</tr>
<tr>
<td>Size (bytes)</td>
<td>32</td>
<td>156</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 16 - Comparison of Pascal Case functions

saved pointer and data from the PX instruction stream, leaving the result in register r1. Because only one opcode is available for calling three micro-coded functions, the microprogram must fetch and decode a one byte sub-op code, following the XFC in the VAX instruction stream.
Pascal-assist extended instructions:

one to do fetch-P-opcode-and-dispatch inside the PX interpreter.

and a pair to help compute L-values, for the LV and RV routines.

The dispatch instruction grabs the next pascal opcode from the location addressed by r10 (which register is incremented after use), and the dispatch table is addressed by r6, which is also added to the offsets thereby fetched, to form the target jump address.

This instruction takes the form:

```
.byte 0xfc
.byte 3
```

Note the constant argument of 3, a sub-opcode. This works very much like a CASEW instruction, but the dispatch table need not follow the instruction, no range checking is done, and the case selector is implicitly the unsigned byte from the PX instruction stream.

At interpreter initialization, one needs to save the display base for the faster forming of LV's. This is accomplished by the code sequence:

```
mov al -display.r0
.byte 0xfc
.byte 0
```

Which saves the contents of r0 in register SSP, the supervisor stack pointer, which UNIX doesn’t use, but which gets saved on a per-process basis by the context-switching instructions.

An lv can thereafter be generated in register r1 by:

```
.byte 0xfc
.byte 2
```

Which picks up a one-byte lex-level and a two-byte (signed) displacement from the PX instruction stream addressed by r10 (which is then incremented), and, using the display-base stored in SSP, forms the L-value (absolute address), which is returned in register r1.

```
begin ; L-value routines
  reg display = SSP ; where we keep it
  reg ll = t0 unsigned byte ; lex-level from PX stream
  reg displacement = t2 signed word ; displacement from PX stream
  reg temp lc = t1
  reg ll_base = t4 ; stack frame base for addressed level
  reg tbl_entry = t3 ; a temporary

fetch_base:
  move display, r0
  exit

lv:
  load ll lc ; get lex level
  add temp lc, lc, 1 ; bump address
  load displacement, temp lc ; get displacement
  sll tbl_entry, ll, 2 ; make lex level address longs in display
  add tbl_entry, tbl_entry, display
  load ll_base, tbl_entry ; fetch display entry
  add r1, ll_base, displacement ; add in displacement
  add lc, temp lc, 2
  exit
end

begin; dispatch instruction
  reg tbl = t8 ; address of address table
  reg opcode = t1 unsigned byte; PX opcode
  reg t_entry = t0 signed word; offset, from table
  reg jaddress = t2
  reg lookup = t3

dispatch:
  load opcode, lc ; fetch PX opcode
  sll lookup, opcode, 1 ; shift op to address words
  add lookup, lookup, tbl ; add table base - now
    points to table entry
  load t_entry, lookup ; fetch displacement from table
  add jaddress, t_entry, tbl ; add to table base
  add lc, lc, 1 ; update PX location pointer
  exit jaddress ; split
end

Figure 17a - PX-assist Case and L-value computation
address | microinstruction
---|---
(10e0): | J/(1404)
(1400): | SUB/1, J/(e84)
(1401): | SUB/1, J/(880)
(1402): | QK/e, BEN/b, IBC/7, J/(1400)
(1403): | SUB/2, IBC/d, J/(1)
(1404): | PKC/4, QK/e, SUB/1, BEN/b, IBC/7, J/(1400)
(1405): | DX/c, J/(1406)

(1406): | BEN/19, J/(140c).
(140c): | SPO/40, ALU/f, AMX/0, DK/8, J/(1407)
(140d): | VAK/1, SPO/4a, ALU/f, AMX/0, J/(1408)
(140e): | VAK/1, SPO/4a, ALU/f, AMX/0, J/(141f)
(140f): | VAK/1, SPO/4a, ALU/f, AMX/0, J/(141f)
(1407): | PKC/4, FS/1, CID/f, KMX/2a, IBC/c, J/(62)
(1408): | FS/0, CID/8, ADS/0, DT/2, J/(1409)
(1409): | SPO/30, KMX/12, ALU/d, RMX/0, AMX/1, BMX/6, J/(140a)
(140a): | SPO/4a, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(140b)
(140b): | SPO/31, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1410)
(1410): | VAK/1, SPO/21, ALU/e, BMX/4, J/(1411)
(1411): | FS/0, CID/8, ADS/0, DT/1, J/(1412)
(1412): | SPO/32, ALU/f, RMX/0, DT/1, AMX/2, J/(1413)
(1413): | SPO/20, QX/8, SI/2, ALU/e, DT/0, BMX/4, J/(1414)
(1414): | SPO/33, FS/1, CID/5, QK/e, KMX/2a, ALU/f, RMX/1, AMX/1, J/(1415)
(1415): | SPO/23, ALU/5, RMX/1, AMX/1, BMX/4, J/(1416)
(1416): | SPO/33, ALU/5, RMX/1, AMX/1, BMX/4, J/(1417)
(1417): | VAK/1, SPO/23, ALU/e, BMX/4, J/(1418)
Figure 17b - VAX microcode generated for Figure 17a
10E0: Q_IB.BDEST, PC_PC+1, IB.TEST?
1400: CALL, J_IB.TBM
1401: CALL, J_IB.ERR
1402: Q_IB.BDEST, IB.TEST?, J/1400
1403: CLR_IB.SPEC, D_Q
D3-0?, VA_R[R10]

= 1100
D_R[R0], J/FBASE
D[BYTE]_CACHE, J/LV
D[BYTE]_CACHE, J/DISP
D[BYTE]_CACHE, J/DISP

= END
FBASE: ID[SSP].D, PC_PC+1, CLR_IB.OPC, J/IRD
LV: ALU_D.OXT[BYTE], D_ALU.LEFT, Q_ID[SSP]
ALU_D, RC[TO]_ALU.LEFT
VA_LA+K[.1]
D[WORD]_CACHE, LC_RC[TO]
VA_ALU, ALU_Q+LC, Q_D
D[LONG]_CACHE
Q.Q.SXT[WORD]
R[R1]_Q+D, DT/LONG
ALU_R[R10]+K[.1]
R[R10].LA+K[.1], DT/LONG, PC_PC+1,
CLR_IB.OPC, J/IRD
DISP: ALU_D.OXT[BYTE], Q_ALU.LEFT,
LAB_R[R8]
VA_LA+Q
D[WORD]_CACHE, ALU_R[R10]+K[.1]
R[R10].LA+K[.1], DT/LONG
LAB_R[R8]
ALU_D.SXT[WORD]+LB,
PC&VA_ALU, FLUSH_IB, J_IB.FILL

Figure 18 - Hand-coded PX-assist

cvtbl (lc)+, r0
cvtwl (lc)+, r1
add12 _display[r0], r1

Figure 19 - VAX assembly code to compute L-value

<table>
<thead>
<tr>
<th>PX-assist</th>
<th>YALLL</th>
<th>hand</th>
<th>assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>size (bytes)</td>
<td>600</td>
<td>300</td>
<td>7 (disp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14 (L-val)</td>
</tr>
<tr>
<td>speed disp</td>
<td>4.8</td>
<td>2.4</td>
<td>4.4</td>
</tr>
<tr>
<td>L-val</td>
<td>4.8</td>
<td>2.5</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Figure 20 - Pascal-assist routine comparisons

A version of the Pascal interpreter actually employing the microcode
routines was timed on two benchmark Pascal programs: finding a solution to the eight-queens problem, and an assignment statement nested in two for-loops. Comparative times, and an indication of the number of times each routine was called are shown in Figure 21.

3. Emulator

The final example is the emulator program of the tutorial in chapter three. This takes forty-two lines of YALLL source and produces fifty-seven VAX microinstructions (Figure 22). The same emulator requires seventy-four hand-written microinstructions on the HPMXE. The number of microinstructions executed to interpret each SM-1 instruction seems to be comparable for the two machines; the extra length of the HP microprogram appears to be due to the necessity of passing parameters in memory on this machine. It should be mentioned that the HP microcode, though vertical and much easier to read than VAX microassembler, is still considerably harder to read and write than is YALLL.

Some crude measurements of emulated instruction times were taken, and are shown in Figure 23. These timings were made by reading the system clock, executing the emulator and reading the clock again. The emulated program executed the instruction under test 36768 times in a loop. Because the system clock only has a ten microsecond resolution, these figures are not very accurate. Instruction counting gives a time estimate of 2.6

<table>
<thead>
<tr>
<th>program</th>
<th>PX with no microcode</th>
<th>microcode assist</th>
<th>number of case jumps</th>
<th>L-values</th>
</tr>
</thead>
<tbody>
<tr>
<td>eight queens</td>
<td>7.9</td>
<td>11.5</td>
<td>717457</td>
<td>247678</td>
</tr>
<tr>
<td>for-loop</td>
<td>15.4</td>
<td>22.8</td>
<td>1518018</td>
<td>508002</td>
</tr>
</tbody>
</table>

Figure 21 - Time in seconds for Pascal programs with and without microcode assist
<table>
<thead>
<tr>
<th>address</th>
<th>microinstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10e0):</td>
<td>SPO/43, QK/8, SI/2, ALU/f, AMX/0, SHF/1, J/(1416)</td>
</tr>
<tr>
<td>(1416):</td>
<td>SPO/44, ALU/5, RMX/1, AMX/1, BMX/3, J/(1416)</td>
</tr>
<tr>
<td>(1418):</td>
<td>VAK/1, ALU/5, RMX/1, AMX/1, BMX/3, J/(1419)</td>
</tr>
<tr>
<td>(1419):</td>
<td>FS/0, CID/8, ADS/0, DT/1, J/(141a)</td>
</tr>
<tr>
<td>(141a):</td>
<td>QK/8, ALU/f, RMX/0, DT/1, AMX/2, J/(141b)</td>
</tr>
<tr>
<td>(141b):</td>
<td>SPO/52, ALU/f, RMX/1, DT/0, AMX/1, J/(141c)</td>
</tr>
<tr>
<td>(141c):</td>
<td>SPO/42, QK/8, KMX/20, ALU/d, AMX/0, BMX/6, J/(141d)</td>
</tr>
<tr>
<td>(141d):</td>
<td>SPO/30, ALU/f, RMX/1, AMX/1, J/(141e)</td>
</tr>
<tr>
<td>(141e):</td>
<td>EALU/3, SMX/0, EBMX/1, SCK/1, SPO/42, KMX/37, ALU/f, AMX/0, SHF/3, DK/8, J/(1420)</td>
</tr>
<tr>
<td>(1420):</td>
<td>DK/d, J/(1421)</td>
</tr>
<tr>
<td>(1421):</td>
<td>BEN/19, J/(1400), 16-way branch</td>
</tr>
<tr>
<td>(1400):</td>
<td>SPO/40, QK/8, ALU/a, AMX/0, J/(1424)</td>
</tr>
<tr>
<td>(1401):</td>
<td>SPO/40, QK/8, SI/2, ALU/f, AMX/0, SHF/2, J/(1425)</td>
</tr>
<tr>
<td>(1402):</td>
<td>CCK/1, SPO/40, ALU/f, AMX/0, J/(1426)</td>
</tr>
<tr>
<td>(1403):</td>
<td>SPO/20, QK/8, ALU/e, BMX/4, J/(1427)</td>
</tr>
<tr>
<td>(1404):</td>
<td>SPO/20, QK/8, SI/2, ALU/e, BMX/4, SHF/1, J/(1428)</td>
</tr>
<tr>
<td>(1405):</td>
<td>J/(1410)</td>
</tr>
<tr>
<td>(1406):</td>
<td>J/(1412)</td>
</tr>
<tr>
<td>(1407):</td>
<td>J/(1414)</td>
</tr>
<tr>
<td>(1408):</td>
<td>SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1422)</td>
</tr>
<tr>
<td>(1409):</td>
<td>SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1422)</td>
</tr>
<tr>
<td>(140a):</td>
<td>SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1422)</td>
</tr>
<tr>
<td>(140b):</td>
<td>SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1422)</td>
</tr>
<tr>
<td>(140c):</td>
<td>SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1422)</td>
</tr>
<tr>
<td>(140d):</td>
<td>SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6, J/(1422)</td>
</tr>
</tbody>
</table>
(140e): SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6. J/(1422)
(140f): SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6. J/(1422)
(1422): PCK/4, SPO/53, SI/2, KMX/1, ALU/5, DT/0, AMX/0, BMX/8, SHF/0, IBC/c. J/(62)
(1423): SPO/40, QK/8, ALU/a, AMX/0. J/(1434)
(1424): SPO/50, ALU/f, RMX/1, DT/0, AMX/1. J/(1417)
(1425): SPO/50, ALU/f, RMX/1, DT/0, AMX/1. J/(1417)
(1426): BEN/1b. J/(1417), 2-way branch
(1417): SPO/43, SI/2, KMX/1, ALU/5, AMX/0, BMX/6. J/(1434)
(141f): SPO/20, QK/B, ALU/e, BMX/4. J/(1427)
(1427): SPO/53, ALU/f, RMX/1, DT/0, AMX/1. J/(10e0)
(1428): SPO/30, ALU/f, RMX/1, AMX/1. J/(1429)
(1429): SPO/44, ALU/f, AMX/0, DK/8. J/(142a)
(142a): SPO/20, ALU/5, RMX/0, AMX/1, BMX/4. J/(142b)
(142b): VAK/1, ALU/5, RMX/0, AMX/1, BMX/4. J/(142c)
(142c): FS/0, CID/8, ADS/0, DT/1. J/(142d)
(142d): QK/8, ALU/f, RMX/0, DT/1, AMX/2. J/(142e)
(142e): SPO/50, ALU/f, RMX/1, DT/0, AMX/1. J/(1417)
(1410): SPO/20, QK/B, SI/2, SUB/1, ALU/e, BMX/4. J/(1435)
(1411): SPO/42, QK/B, ALU/f, AMX/0. J/(142f)
(142f): SPO/50, ALU/f, RMX/1, DT/0, AMX/1. J/(1417)
(1412): SPO/20, QK/8, SI/2, SUB/1, ALU/e, BMX/4. J/(1435)
(1413): SPO/42, QK/8, ALU/f, AMX/0. J/(1430)
(1430): SPO/40, QK/B, ALU/d, RMX/0, AMX/0, BMX/7. J/(1431)
(1431): SPO/50, ALU/f, RMX/1, DT/0, AMX/1. J/(1417)
(1414): SPO/20, QK/8, SI/2, SUB/1, ALU/e, BMX/4. J/(1435)
(1415): SPO/42, QK/8, ALU/f, AMX/0. J/(1432)
**Figure 22 - VAX microcode for emulator of Chapter three**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>time (usecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>complement</td>
<td>3.1</td>
</tr>
<tr>
<td>shift</td>
<td>3.1</td>
</tr>
<tr>
<td>load</td>
<td>4.6</td>
</tr>
<tr>
<td>store</td>
<td>3.7</td>
</tr>
<tr>
<td>add</td>
<td>4.3</td>
</tr>
<tr>
<td>and</td>
<td>4.3</td>
</tr>
<tr>
<td>jump</td>
<td>2.7</td>
</tr>
<tr>
<td>branch on neg</td>
<td></td>
</tr>
<tr>
<td>successful</td>
<td>3.1</td>
</tr>
<tr>
<td>unsuccessful</td>
<td>2.7</td>
</tr>
</tbody>
</table>

**Figure 23 - VAX SM–1 instruction times**

... microseconds for the emulated jump instruction.
CHAPTER 5

CONCLUSIONS AND COMMENTS

This project has involved many areas of computing: architecture, implementation, operating systems and compiler writing, microprogramming, and of course, documentation. Several of these things seem peripheral to the objective of microprogramming the VAX, but are necessary to realize it. This chapter contains some conclusions drawn from this work, including comments on the VAX microarchitecture, the YALLL language and translator, the translator writing process, and a list of things which need still to be done to create a usable microprogramming environment on the VAX.

1. VAX — Architecture and Implementation

The architecture of the VAX is very well thought out; the variable-length instructions and addressing modes are an evolutionary step upward from the PDP-11 that preceded it, and the variety of instructions provided is unprecedented. But there are two problems with this machine which might have been avoided. The first is complexity: some of the macro-instructions require microprograms which are enormously complicated. This makes the instruction set microprogram noticeably bigger, less understandable, and much more susceptible to error. As an example, there is a bug in the \textit{index} instruction which causes the machine to hang; the routines supporting the string and decimal instructions are huge, and are also known to contain errors, though not so serious as the one in \textit{index}. One way of dealing with such a large microprogram would be to use a high-level language to make the program more understandable. (Presumably, many engineering and
marketing considerations went into the choice of instructions, so that reducing its size is not an alternative in the present design.)

The second problem area is the way in which user microcoding is integrated into the architecture: the XFC instruction causes a trap (exception), and after inspection of a word in the SCB, microprogram execution may branch to user microcode. This accounts for the large (3.5 u-sec) penalty incurred by entering user microcode. The XFC instruction is conceptually not the only way to enter user microcode; any exception or interrupt can enter it (see Appendix A, Figure 5). The problem here is that no matter what causes the user's microcode routine to be executed, that routine can be entered at only one point — location 10E0. If there is more than one way to get there, the user's code will have to determine how it was entered. For future VAXes, I would like to propose one implementation change and one architectural change. First, that there be separate entry points to the user's microcode for exceptions and for interrupts. The trap-handling microroutine currently keeps a flag in the STATE register indicating interrupt or exception when the branch to 10E0 is taken; this could easily be made a jump to 10DE or 10DF, for instance, conditional on that bit. This would cost no execution time, and only the extra one word of space. I further propose that the XFC instruction be handled separately from the exception mechanism, and given its own user microcode entry point, entered immediately upon recognition of the op-code. In the (default) case where no user microcode is loaded, this could then branch into the exception handling, and be treated as an illegal instruction. This would cause its recognition as an illegal op-code to be about 0.2 u-sec slower than other illegal instructions, but entry to user's microprograms would be 3 microseconds faster than at present.
This machine is among the most complex of user-microprogrammable computers, rivaling the QM–1, because of its great microinstruction width, the special-case optimizations for various operations, and the complexity of devices such as the accelerator, instruction buffer, translation buffer, and cache. Actually, the latter two are extremely easy to use under most circumstances, but one must always be aware of their presence. The instruction buffer is more complicated to use, and although fetching byte operands seems fairly straightforward, it has not yet been discovered if this device can be employed by a user microprogram to execute more complex functions, such as operand specifier decoding.

The central data paths are quite complex, and are very well suited to executing VAX and PDP–11 instructions. This is one reason it is so hard to write microcode which can out-perform a short sequence of native instructions.

2. YALLL – Language and Implementation

The YALLL language has been implemented on two machines: the VAX and the HP 300. One of its advantages is supposed to be the transportability of YALLL microprograms. While it is true that YALLL programs written for one machine can be compiled and run on the other, the sense of the program might not so easily be transferred. Such things as a machine’s parameter passing conventions or addressing scheme can determine the environment in which the microprogram runs, and what sorts of operations it needs to perform to deal with the macroarchitecture. The choice of two such different machines accentuated this problem, and it is felt that transporting a microprogram between more similar computers (such as different models of a VAX family) would yield much more satisfying results.
Counterbalancing this small shortcoming, YALLL has one great strength — it makes microprogramming the VAX doable. YALLL is very much easier to read and write than VAX microassembler, even with macros. Hand microprogramming may still be necessary for routines which must be small and fast, or deal with special resources, but even these should be debugged using YALLL. The microprogramming tools to be offered by DEC do not look that appealing: a macro-assembler and a VAX-to-control store loader. The macro-assembler will make programs less tedious to write, but one will still have to understand the intricacies of the VAX data paths in great detail. And the loader does not write a control store image on floppy disk, but writes directly to control store. Thus the microprogrammer will not be able to use the LSI–11's microdebugger to interactively alter his microprogram. This scheme does have the advantage that it is not necessary to halt the VAX to load a microprogram, but prudence dictates that it be stopped anyway, at least when a microprogram is in the debugging stage.

There are several problems with YALLL/VAX which I feel are more implementation problems than language design problems. The first is that it does not produce very efficient code. Hand coding a routine generally results in a program which is half as long and twice as fast. The second problem is the binding of variables to registers. These bindings are taken quite literally by the translator, so that a move from a variable bound to register r1 will always cause this register to be accessed, even if it was just loaded from the easier-to-access Q register, which thus contains a copy of it. It is this problem which has caused me to observe that the VAX is really a two-register machine (the D and Q registers) with some fast local store (RAB and RC). A related problem is the type mechanism, which is the occasional cause of extra transfers, to sign- or zero-fill quantities. The current typing and conversion
system is not especially well thought out, and ought to be changed.

These are all problems which could easily have been dealt with had we not been forced to spend so much time demystifying the microarchitecture. Had we been handed a document like Appendix A six months ago, we would have been able to take much more care in the construction of the code generation routines. Given the circumstances, we are happy with the results.

3. Reflections on the Program Development Process

One of the most thought-provoking aspects of this project was the project itself, seen as an exercise. From this experience, I learned quite a few things about the development of medium-sized software, and now know several things not to repeat, but mainly a lot of programming techniques that worked quite well.

The first lesson is that documenting a machine and writing a translator for it are really two separate projects, and do not complement each other too well. This documentation task was not one of the project's original objectives, but these objectives were formed before the VAX was available, and before we could assess the lack of documentation supplied by the manufacturer. As of this writing (July '79), the promised data path description has yet to be seen. An allied problem is that documenting a heavily-used machine is difficult and inconvenient. Because of the demands on the VAX's time, I estimate that I have spent much fewer than forty hours of stand-alone time investigating the microprogramming.

The writing of the translator, on the other hand, went very well, largely due to the software tools available – the Unix editors, C language, Lex and YACC. As Brooks points out [Brooks 75]
Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include.

He concludes that "The most important two tools for system programming today . . . are (1) high-level language and (2) interactive programming."

Tools such as YACC and Lex increase productivity even more, since they automate the writing of two not-very-interesting parts of a translator, the scanner and parser. This makes it quite reasonable to change to language well after the translator is begun.

A programming practice which was found to be extremely useful was to include internal checks in many areas of the program. The symbol table and lowest level code generation routine, for instance, check the validity of their arguments and the consistency of the data structures before proceeding to search through or modify those structures. The data structure representing the generated code is particularly difficult to maintain, and often failed consistency checks, especially during the debugging of the second pass. I am especially happy with these checking facilities, and with those to provide dumps of the symbol and code table, as described in chapter three.

Nearly as important as these internal checks are external checks – the dump programs which insure that the binary file produced by yc and merger are of the correct form. Because of all this self-checking, the microcode tested on the VAX has been surprisingly trouble-free. This is not to claim that it is bug-free, but that each of these bugs seems always to be a single field set to a wrong value, rather than mangled program logic.

4. Things Still to be Done

There are still several things to be done to make microprogramming the VAX-11/780 an easier task. Some are listed here; the first two are real
necessities, the others are ranked in more-or-less descending priority. The most pressing need is for more complete documentation; there are several areas of the VAX microarchitecture which I was not able to investigate sufficiently. For example, it is understood that when doing arithmetic (carry-borrow) operations with certain sources, one should allow an extra cycle before using the result. One DEC employee suggests that this is the case for "slow" constants, from the constant ROM. Someone else says that this is necessary when routing a general register contents through a latch and through the ALU in the same instruction. The YALLL translator emits conservative code to handle both cases, but this may be unnecessary. It is not known if there are timing requirements for any other operations. Perhaps DEC's data path description will clarify this whole area. A further mystery is the use of the instruction buffer (IB) either to fetch multiple byte operands from the instruction stream, or to decode operand specifiers.

A micro-engine simulator would greatly have eased the debugging of the YALLL translator, and is still necessary to ease the debugging of user microprograms. Although the console microcode debugger is an excellent facility, it requires that the machine be used stand-alone. Furthermore, a simulator could write trace information into a file for later analysis and display, rather than forcing the user to single step the machine, then request by name each register to be displayed.

If the YALLL/VAX translator is to be seriously used, it should be rewritten to produce better code. Given the greater information now available on the VAX, it should be reasonably easy to change yc to produce code within 25% of hand-written in both time and space. Simply keeping track of the contents of the D and Q registers should make most of the difference. Since
the thirty-two-bit ALU seems to be the primary bottleneck in YALLL programs, the user of this resource should also be improved. For example, the operation "add Dreg,r0,16" is certainly a two-step operation. The code currently generated for this statement forms the sum in the ALU during two microinstructions, but only gates it into the D register during the second. A better sequence would be to route the constant to BMX and to latch the contents of register r0 into latch LA in one instruction, then form and gate the sum in the next instruction. This still takes two instructions, but only uses the ALU in the second one; the first might now more easily be combined with its predecessor, even if that instruction used the ALU (but not a constant or LA).

One objective of the YALLL programming language is to hide a machine's peculiarities; for example, the way in which shifting is done. However, for machine-specific programming tasks, such as instruction set implementation, it would be desirable to use a high-level microprogramming language which would allow the programmer to exploit a machine's peculiarities, similar to PL/360 [Wirth 68]. For example, YALLL has six shifting operators; aside from special cases, the VAX has one — a double shift with single-word result. An operation such as "Dreg <-- (A,B)shift(C)" could load A into register Q, B into D, C into SC, do the shift, and leave the result in the D register. The YALLL programmer has no way to specify use of the EALU, or to hint at possible parallelism to the compiler; both these facilities could lead to better microprograms. Finally, to interact fully with the macroarchitecture, a microprogrammer must be able to specify the length of a result to be loaded into a general-purpose register. For the reasons previously outlined, YALLL programs always load longword results into the registers; unfortunately, this is not fully compatible with the VAX architecture.
Originally, YALLL was meant to be a low-level intermediate language, to be used as the output of a higher-level language, such as Modula [Wirth 77]. Now that YALLL is implemented, Modula can be modified to make it a reasonable microprogramming language (eg: addition of memory as a pre-defined object) and an M-code to YALLL translator can be written. Such a program would have the additional advantage that it could easily be written to emit code for any computer having a YALLL compiler.

Finally, a DEC-compatible macro-microassembler should be implemented under Unix, so that one could use the macro-facility used by the VAX implementors. Perhaps the assembler-escape mechanism in YALLL/VAX (which is currently very simple) could be rewritten to recognize their symbols and macros. This facility is not really necessary, but when microprogramming this machine, one needs all the help one can get.
BIBLIOGRAPHY


[Bondy 77] Bondy, J.L., and Freeman, D.N. “Putting Supervisory Routines into Hardware” Proceedings of the IFIP 1975


[Lesk 75] Lesk, M.E. “Lex – A Lexical Analyzer Generator” Bell Labs, Murray Hill, N.J. 1975

Microprogramming Workshop Hershey, Pa. 1979


APPENDIX A

VAX ARCHITECTURE

1. OVERVIEW

In order to successfully microprogram any computer, one must understand the underlying design, especially when that design is as full of peculiarities and optimizations as is the micro-architecture to the VAX-11/780. In this paper, I shall discuss that machine's architecture, and how it relates to the writer of "user" micro-code - that not supporting the machine's inherent instruction set.

Before one can make sense out of the low-level design, one must be familiar with the high-level architecture it is designed to support. For that reason, I will first examine some of the features of the VAX's macro-instruction set which are reflected in the design of the micro-machine. In particular, I shall discuss the instruction format, addressing modes, data types, and support for memory management and the operating system. The reader familiar with these features might thus skim the first section.

1.1. MACRO-LEVEL ARCHITECTURE

1.1.1. PMS Structure

The PMS structure of the VAX-11 differs substantially from that of any of the PDP-11 series of computers; all use a bus (or busses) shared by Pc, Mp, Ms, and T's, but there the similarity ends. The VAX uses a hierarchy of busses, of which the primary one is the synchronous backplane
interconnection (SBI). This bus has a thirty-two bit wide data and address path and a 200 nano-second cycle. The subsidiary Unibus is the same as that used by PDP-11 computers, so Unibus peripherals may be connected to it. The Unibus has eighteen bit addresses and a sixteen bit data path; the Massbus has a thirty-two bit data path. Both K(Unibus) and K(Massbus) are capable of mapping twenty-eight bit SBI addresses into bus addresses, using simple memory map mechanisms contained in the controllers. I/O devices are constrained to the top half of the SBI's giga-byte address space, and all Mp addresses are in the lower 512 mega-bytes. The PMS structure is illustrated by Figure 1.

1.1.2. the CPU

1.1.2.1. registers, instruction format

The VAX has sixteen, thirty-two-bit "general" registers, and many special-purpose control registers. The general register layout is similar to that of the PDP-11, in that one of the registers (#15) is really the program counter, PC, and another (#14) is the stack pointer, SP. Some of the other registers are appropriated by the string handling and subroutine call instructions.

The instruction format is variable - a one-byte opcode followed by up to six operands or addresses (each of which is known as an operand specifier). Memory is byte addressable. The addressing scheme is a logical extension to that used by the PDP-11; operand addresses always consist of at least one byte, of which four bits determine the addressing mode, and four bits designate one of the general registers, which is to be used in address formation. The primary addressing extensions concern displacement addressing, a new
immediate mode, and a new indexing scheme. The addressing modes are summarized in Figure 2.

1.1.2.2. addressing modes

In register mode, the required datum resides in the general register designated in the specifier byte, or that register and the next, for operands

---

1 DEC asmler notation. Unix assembler differs.
<table>
<thead>
<tr>
<th>NAME</th>
<th>NOTATION¹</th>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>literal</td>
<td>S~#literal</td>
<td></td>
</tr>
<tr>
<td>indexed</td>
<td>i[Rx]</td>
<td>(rest of addr) 4 n</td>
</tr>
<tr>
<td>register</td>
<td>Rn</td>
<td>5 n</td>
</tr>
<tr>
<td>register deferred</td>
<td>(Rn)</td>
<td>6 n</td>
</tr>
<tr>
<td>autodecrement</td>
<td>-(Rn)</td>
<td>7 n</td>
</tr>
<tr>
<td>autoincrement</td>
<td>(Rn)+</td>
<td>8 n</td>
</tr>
<tr>
<td>autoincrement deferred</td>
<td>@(Rn)+</td>
<td>9 n</td>
</tr>
<tr>
<td>byte displacement</td>
<td>B~d(Rn)</td>
<td></td>
</tr>
<tr>
<td>byte displacement deferred</td>
<td>@B~d(Rn)</td>
<td>d A, n</td>
</tr>
<tr>
<td>word displacement</td>
<td>W~d(Rn)</td>
<td></td>
</tr>
<tr>
<td>word displacement deferred</td>
<td>@W~d(Rn)</td>
<td>d C, n</td>
</tr>
<tr>
<td>longword displacement</td>
<td>L~d(Rn)</td>
<td></td>
</tr>
<tr>
<td>longword displacement deferred</td>
<td>@L~d(Rn)</td>
<td>d E, n</td>
</tr>
<tr>
<td>immediate</td>
<td>i~#const</td>
<td>const</td>
</tr>
<tr>
<td>absolute</td>
<td>@#const</td>
<td>const</td>
</tr>
<tr>
<td>byte relative</td>
<td>B~d</td>
<td>d A, F</td>
</tr>
<tr>
<td>byte relative deferred</td>
<td>@B~d</td>
<td>d B, F</td>
</tr>
<tr>
<td>word relative</td>
<td>W~d</td>
<td>d C, F</td>
</tr>
<tr>
<td>word relative deferred</td>
<td>@W~d</td>
<td>d D, F</td>
</tr>
<tr>
<td>longword relative</td>
<td>L~d</td>
<td>d E, F</td>
</tr>
<tr>
<td>longword relative deferred</td>
<td>@L~d</td>
<td>d F, F</td>
</tr>
</tbody>
</table>

Figure 2 - Address modes

The table above lists various name notation for different address modes. Each address mode is associated with a specific notation and format. For example, the literal notation is `S~#literal` and the format is `0 l lit` for 32-bit signed immediate. Each notation is used to specify how an address is formed, with the format indicating how the address is stored in memory. The notation includes various components such as register, displacement, and immediate values.

A register deferred mode address is one in which the designated register contains the address of (pointer to) the desired operand. Autodecrement mode is similar, but in this case, after the register's contents have been used to find the required operand, they are augmented by the length of that operand, in bytes. Thus the "C" language idiom "*p++" can be directly implemented, for register variable p, with this addressing mode. Using auto-increment deferred mode, the register contents address not the
datum, but another pointer, addressing the datum. The register is incremented by four (the size of the addresses address) after use.

Auto-decrement mode is similar to auto-increment, but here, the register is decremented by the length of the operand before it is used as an address. This symmetry provides the stack operations push and pop. There is no auto-decrement deferred.

Immediate mode, coded as auto-increment specifying the PC as register, provides for the required datum, be it byte, word, longword, or quadword, to follow the specifier byte directly in the instruction stream. Needless to say, such an operand cannot be used as an operator's destination. The new literal mode is a further method for specifying short instruction-stream data. Normally, an instruction such as MOVL #const, dest will take the form:  

\[
\text{dest specifier} \quad \text{const} \quad 8F \quad \text{DO} \\
\begin{array}{c}
\text{const} \\
\text{spec} \\
\text{opcode}
\end{array}
\]

However, if the constant is small (zero through sixty-three, inclusive, or certain select floating-point quantities), the four bytes of constant can be saved, and the datum placed in the mode specifying byte.

Absolute mode (autoincrement deferred specifying PC) provides for the four-byte absolute address of the operand to follow the address specifier directly in the instruction stream.

Most machines which can use base + displacement addressing allow a fixed number of bits for the displacement (e.g. sixteen bits on the PDP-11, twelve bits for the IBM-360). This is very often excessive, as when addressing

---

\[2\] By VAX convention, memory-byte addresses increase from right to left.
small data structures via a base pointer in a register. Alternatively, a displacement smaller than the logical address space may be insufficient; addressing large arrays on the IBM-360 may require multiple base registers. The VAX architects attacked this problem by allowing three displacement field sizes: byte, word and longword, for three displacement addressing modes. Another set of modes, the displacement deferred modes, use the base + displacement address to point to a pointer, of four bytes. By specifying the PC as "base" register, the relative and relative deferred modes are obtained, the inference being that the datum (or a pointer to it) resides at an address formed from the address of the specifier byte, and the byte, word, or longword displacement which follows.

The most interesting mode is the new index addressing mode. Actually, this must be combined with another mode, such as base + displacement; or absolute. Unlike the IBM-360, in which the address is the sum of the base register, displacement field, and the index register, VAX indexed addresses are formed as a starting address (which may be base register plus displacement field) added to the product of the index with the length of the datum being addressed, be it one, two, four, or eight bytes. So a loop index can be used directly for indexing into a vector.

1.1.2.3. data types

A striking feature of the VAX instruction set is the plethora of data types supported; the two's complement integer types are byte, word (two bytes), longword (four bytes), and quadword (eight bytes). Floating point numbers are represented by a single precision type (sign; eight bit exponent, excess 128; twenty-three bit fraction), and by a double precision type (sign; eight bit
exponent; fifty-five bit fraction). Packed decimal numbers of up to thirty-one
digits in length are also supported. A full range of arithmetic instructions is
included for all of these types, excepting quadword:

\[
\begin{align*}
\text{move} & \quad \text{byte} \\
\text{compare} & \quad \text{word} \\
\text{add} & \quad \text{longword} \\
\text{subtract} & \quad \text{floating} \\
\text{multiply} & \quad \text{double floating} \\
\text{divide} & \quad \text{packed decimal}
\end{align*}
\]

In addition, instructions exist to convert any of the floating and binary types
to any other (excepting quadword). Decimal types can only be converted to
and from longword integers, and various character formats. Quadword
integers are not fully supported; in fact, they might not be considered a
separate data type on this machine, but a case of multiple-precision
integers. Multiple-precision arithmetic is aided by longword add and sub-
tract instructions which use the carry generated by a previous operation.

The decimal instructions appear to be DEC's gesture towards the busi-
ness market, as do the suspiciously IBM-like decimal-character conversion
instructions. Two instructions convert from packed decimal to "trailing
numeric" character strings and back, and two convert between packed
decimal and "leading separate numeric" strings. Finally, there is the EDIT
instruction. This is very much like the IBM-360's instruction for converting
packed decimal numbers to punctuated character strings, but has several
more pattern characters. Furthermore, it is a triple address instruction -

---

No 2-operand decimal multiplication or division. Binary types are further supported by
operations not shown here.

Packed decimal strings contain two BCD digits per byte, and the low-order four bits of the
low-order (highest addressed) byte contain the sign. Trailing numeric numbers use the ASCII
representations of each digit excepting the low-order digit, in which the sign is also encoded. In
leading separate numeric format, a separate byte containing a representation of the sign
precedes the ASCII digit bytes.
the edit pattern is not overwritten by the edited string. Conspicuous in their absence are any instructions to convert between character strings and binary integers or floating point numbers.

Although character strings and bit fields within a word might not be classed as "data types", the VAX does provide for some manipulation of them. Character strings may be moved, compared with one another, translated, scanned or spanned (using a single character pattern or table), and searched for substrings. Bit fields can be inserted or extracted from memory, or compared with one another.

1.1.2.4. memory management

The VAX logical address space is divided into system space and user space; user space is in turn divided into two regions (which DEC refrains from calling segments). These three regions (system, P0, P1, which grows backwards to accommodate the user's stack) are further divided into 512-byte pages. Because the system and the user co-exist in the same addressing space, moving data between them is simplified. For each active page in the logical space, there is a longword page table entry (PTE). If a single page table were used to map both user and system regions, it would have to be over 4k bytes long, just to attain the lowest system address. To yield the page table more compact, three tables are actually used. For each of the regions P0, P1, and system, there is a page table origin register, and an associated table length register. The system table is addressed in absolute memory, and the user tables are in system virtual memory. To speed memory references by the CPU, a cache and a translation buffer (TB) are used.
Figure 3 - Address Space

a) Logical

- $P_0$
  - grows forward
  - $4000\,0000_{16}$
- $P_1$
  - grows backward
  - $8000\,0000_{16}$
  - grows forward
  - $C000\,0000_{16}$
  - reserved

b) Physical

- Memory Controller 1
  - (1 or 2 M bytes)
  - $0000\,0000_{16}$
- Memory Controller 2
- Memory Address Space
- Device Address Space
- UNIBUS Address Space

*One of 16, 8-K-byte Regions
†One of 4, 256-K-byte Regions
### Figure 4a - Page Table Entry

<table>
<thead>
<tr>
<th>V</th>
<th>valid</th>
<th>0 not in MP</th>
<th>1 Mp resident</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>modified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OWN</td>
<td>software use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PFN</td>
<td>page frame number</td>
<td>upper 21 bits of physical address, if V=1</td>
<td></td>
</tr>
</tbody>
</table>

#### PROT - protection

<table>
<thead>
<tr>
<th>K</th>
<th>E</th>
<th>S</th>
<th>U</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>rw</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>r</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
<tr>
<td>5</td>
<td>rw</td>
<td>rw</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>rw</td>
<td>r</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>r</td>
<td>r</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>r</td>
</tr>
<tr>
<td>10</td>
<td>rw</td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>11</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>12</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
<td>r</td>
</tr>
<tr>
<td>13</td>
<td>rw</td>
<td>rw</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>14</td>
<td>rw</td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
</tbody>
</table>

- **K** - Kernel access
- **E** - Executive access
- **S** - Supervisor access
- **U** - User access

- **-** - no access
- **x** - unpredictable
- **r** - read
- **rw** - read/write
1.1.2.5. interrupts, other operating systems considerations

The VAX, as befitting a computer designed in this age of hierarchical software systems, has four execution modes of increasing privilege: user, supervisor, executive, and kernel (of which UNIX uses only user and kernel). Each mode has its own stack pointer; USP, SSP, ESP, KSP. Additionally, there is an "interrupt stack pointer" (ISP) for use in kernel mode. The processor exchanges stack pointers whenever the mode changes. Thus, for example, a supervisor state program does not have to be concerned about
using a stack which the user program may have caused to overflow; one should never trust code of lower privilege than oneself. The stack pointers, excepting ISP, are considered to be per-process registers, and are swapped during process switching.

When a program interrupt or exception is taken, the processor handles much of the state saving and switching. First, a longword is taken from the system control block, whose physical base address is kept in register SCBB. The word taken from this block depends on the cause of the interruption (see Figure 5). This word is interpreted thus: bits <1:0> determine the action to be taken; bits <31:2>, catenated with "00" on the right, may be used as an interrupt address. If the action code is "10", control is given to a user-written (i.e. not DEC-supplied) micro-routine at location 10E0(hex) in control store. Otherwise, KSP (on code "00") or ISP (on "01") is selected as the new stack pointer, interrupt information is pushed on the stack (including the PSL, PC, and, in some cases, an exception code), and kernel mode instruction execution resumes at the interrupt address. If the action code is "11", the information is pushed on the kernel stack, and the processor halts.

All the per-processor registers (including the general and memory map registers) can be saved or restored in one instruction execution, so context switching is potentially very fast. The registers are saved or restored from a "process control block" (as shown in Figure 6) whose physical address is contained in register PCBB.
### SYSTEM CONTROL BLOCK

<table>
<thead>
<tr>
<th>Offset</th>
<th>Interruption or Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unused</td>
</tr>
<tr>
<td>4</td>
<td>machine check</td>
</tr>
<tr>
<td>8</td>
<td>kernel stack not valid</td>
</tr>
<tr>
<td>C</td>
<td>power fail</td>
</tr>
<tr>
<td>10</td>
<td>reserved or privileged instruction</td>
</tr>
<tr>
<td>14</td>
<td>XFC</td>
</tr>
<tr>
<td>18</td>
<td>reserved or illegal operand</td>
</tr>
<tr>
<td>1C</td>
<td>reserved or illegal address mode</td>
</tr>
<tr>
<td>20</td>
<td>access violation</td>
</tr>
<tr>
<td>24</td>
<td>translation not valid (page fault)</td>
</tr>
<tr>
<td>28</td>
<td>trace trap</td>
</tr>
<tr>
<td>2C</td>
<td>breakpoint trap</td>
</tr>
<tr>
<td>30</td>
<td>compatibility mode trap</td>
</tr>
<tr>
<td>34</td>
<td>arithmetic trap</td>
</tr>
<tr>
<td>38</td>
<td>unused</td>
</tr>
<tr>
<td>3C</td>
<td>unused</td>
</tr>
<tr>
<td>40</td>
<td>change mode to kernel</td>
</tr>
<tr>
<td>44</td>
<td>change mode to executive</td>
</tr>
<tr>
<td>48</td>
<td>change mode to supervisor</td>
</tr>
<tr>
<td>4C</td>
<td>change mode to user</td>
</tr>
<tr>
<td>50 - 80</td>
<td>unused</td>
</tr>
<tr>
<td>84</td>
<td>software level 1</td>
</tr>
<tr>
<td>88</td>
<td>software level 2</td>
</tr>
<tr>
<td>8C - BC</td>
<td>software levels 3 - F</td>
</tr>
<tr>
<td>C0</td>
<td>interval timer</td>
</tr>
<tr>
<td>C4 - F4</td>
<td>unused</td>
</tr>
<tr>
<td>F8</td>
<td>console terminal receive</td>
</tr>
<tr>
<td>FC</td>
<td>console terminal transmit</td>
</tr>
<tr>
<td>100</td>
<td>device level 14, device 0</td>
</tr>
<tr>
<td>1FF</td>
<td>device level 17, device 15</td>
</tr>
</tbody>
</table>

---

**SCB ENTRY**

<table>
<thead>
<tr>
<th>SCB ENTRY</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op: 0</td>
<td>use KSP, unless already on ISP</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>use ISP, on exception IPL raised to 1F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>micro-branch to 10E0 in WCS</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 5 - System Control Block**
### PROCESS CONTROL BLOCK

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KSP</td>
</tr>
<tr>
<td>4</td>
<td>ESP</td>
</tr>
<tr>
<td>8</td>
<td>SSP</td>
</tr>
<tr>
<td>C</td>
<td>USP</td>
</tr>
<tr>
<td>10</td>
<td>r0</td>
</tr>
<tr>
<td>14</td>
<td>r1</td>
</tr>
<tr>
<td>18</td>
<td>r2</td>
</tr>
<tr>
<td>1C</td>
<td>r3</td>
</tr>
<tr>
<td>20</td>
<td>r4</td>
</tr>
<tr>
<td>24</td>
<td>r5</td>
</tr>
<tr>
<td>28</td>
<td>r6</td>
</tr>
<tr>
<td>2C</td>
<td>r7</td>
</tr>
<tr>
<td>30</td>
<td>r8</td>
</tr>
<tr>
<td>34</td>
<td>r9</td>
</tr>
<tr>
<td>38</td>
<td>r10</td>
</tr>
<tr>
<td>3C</td>
<td>r11</td>
</tr>
<tr>
<td>40</td>
<td>r12 (ap)</td>
</tr>
<tr>
<td>44</td>
<td>r13 (fp)</td>
</tr>
<tr>
<td>48</td>
<td>r15 (pc)</td>
</tr>
<tr>
<td>4C</td>
<td>PSL</td>
</tr>
<tr>
<td>50</td>
<td>POBR</td>
</tr>
<tr>
<td>54</td>
<td>ASTLVL &lt;26:24&gt;, POVR &lt;21:0&gt;</td>
</tr>
<tr>
<td>58</td>
<td>P1BR</td>
</tr>
<tr>
<td>5C</td>
<td>PME &lt;31&gt;, P1LR &lt;21:0&gt;</td>
</tr>
</tbody>
</table>

**PSL**

<table>
<thead>
<tr>
<th>CM</th>
<th>TP</th>
<th>FP</th>
<th>IS</th>
<th>CMD</th>
<th>PMD</th>
<th>0</th>
<th>DV</th>
<th>FU</th>
<th>IV</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
</table>

- **CM** - compatibility mode
- **TP** - trace pending
- **FP** - first part done
- **IS** - interrupt stack
- **CMD** - current mode (0-kernal; 1-exec; 2-super; 3-user)
- **PMD** - previous mode
- **IPL** - interrupt priority level
- **DV** - decimal overflow trap enable
- **FU** - floating underflow trap enable
- **IV** - integer overflow trap enable
- **T** - trace trap enable
- **N** - negative condition code
- **Z** - zero condition code
- **V** - overflow condition code
- **C** - carry condition code

*Figure 8 - PROCESS CONTROL BLOCK*
1.2. THE CPU

The VAX CPU cannot be viewed as a single machine, but rather as a collection of tightly-coupled units: the instruction buffer, or I-box; the memory management hardware, including the cache and translation mechanisms; the optional floating-point accelerator; and the central, micro-programmed part of the CPU, which I call the Central Data Paths.

1.2.1. the Instruction Buffer

The instruction buffer has the task of fetching the instruction stream from memory, and decoding the instruction op-code and data specifiers (addressing modes). Eight bytes of instruction stream can be accommodated at one time, and if this space is half occupied, or less, the unit will attempt to pre-fetch the next longword of program. A separate address register (IBA) is used for this purpose, and is updated by the I-box. The pre-fetching action can be inhibited by the central microprogram as, for example, when a branch or context switch is about to take place. The buffer can also be cleared, IBA reloaded, and fetching reinitiated in such a case. The possibility of a program's modifying an already-fetched instruction byte dictates the operating system's enforcement of pure procedures.

When interpreting a macro-instruction, the microprogram must fetch and store instruction operands, using the VAX's elaborate addressing scheme. Although total hardware support for this function is not provided, the I-box does give considerable assistance. At least once during the decoding of each macro-instruction, and usually once per operand, the microcode executes a "decision point" (or SPEC) branch, a table jump in which the lowest eight address bits are supplied by the instruction decode logic. The
Each instruction buffer byte has a validity bit indication whether it contains good data. As opcodes and specifiers are evaluated, bytes are shifted towards byte 0. Memory data is rotated according to the two low order bits of register IB4, and loaded according to the validity bits.

FIGURE 7 - the Instruction Buffer
microcode thus executed can then fetch or store the operand, interpret an operand address, cause a reserved address mode fault (e.g., -(PC) mode), or execute the instruction, as in the case of instructions with no explicit operands (like "rei"), or certain operations with some or all operands in registers, (like "addl2 r1,r2"). Branch-on-condition-code instructions are also handled in this manner, as the condition codes and the branch conditions can be compared by the hardware at the first decision point. A three-bit counter called the execution point counter keeps track of the number of decision point branches taken thus far in the interpretation of the current instruction, and so contains the index of the operand being decoded at any time.

A major constituent of the instruction decode logic is a ROM which is addressed two-dimensionally; by opcode, and by operand number, as supplied by the execution point counter. For each operand of every operator, this ROM contains a twelve-bit control word, containing information about the expected operand. This information includes operand size (byte, word, long-word, quadword), type (integer; floating; memory address for string and decimal instructions; memory address or register, for bit field instructions), access necessary (branch displacement, read, write, modify), and four bits of (micro-) address information.

1.2.2. Memory Management Hardware

In a paged environment, every successful memory reference can potentially cause two accesses - one to read the page table, and one to do the prescribed operation. (Since the user page tables are in virtual memory, too, VAX user overhead could be worse.) To speed memory mapping, translation table entries are cached in a translation buffer (TB). The translation
associative with two sets, and holds 128 entries of twenty-one bits plus three bits of parity. The sets are directly addressed by virtual address bits 31 and \( <13:9> \), and deliver physical address bits \( <29:9> \) (this allows mapping of I/O device registers, too). Since bit 31 of a virtual address is used to look up a set of entries, the systems is not competing with the user for TB space. Thus if control is taken from the user by the system program (as for the servicing of an interrupt) and then returned, the user's TB entries will still be intact, and he will not have the overhead of building them back up. (Note, though, that the memory cache does not work this way - allowing the user all the cache entries, and for him to loose many of them during interrupt servicing.) Whenever a CPU data reference causes a TB miss, the microprogram is interrupted, and a micro-routine entered to fetch the missing page table entry into the buffer. This can require multiple fetches, since the user map is in virtual space, too. If the PTE is invalid, of course, the page is missing, and a macro-program interrupt must be taken. The translation mechanism is controlled by a bit in control register MAPEN (ID bus register TBERO). When it is disabled (bit is zero) the low-order bits of longword addresses are used as SBI (physical) addresses.

Another method used to speed memory references is to keep recently used, and what one hopes are soon-to-be used, program and data bytes in a high-speed cache. This is also set associative with two sets, contains long-word entries, and fetches two longwords at a time from memory. It contains 8k bytes, the two-way sets being directly addressed by bits \( <11:2> \) of the physical address. The microprogram assumes the cache always contains the desired data. When there is a cache miss on a data reference, the microprogram is stalled (forced to execute no-ops) until the reference can be
the CPU, main memory is changed correspondingly. This is kept from delaying the processor by a write buffer. If a cached entry in main memory is modified by I/O activity, the corresponding cache entry is invalidated; references to it by the CPU will cause an actual memory access.

An addressing constraint found on IBM-360 computers was that data must fall on "natural" boundaries (halfwords on even addresses, words on multiples of four, etc.). This causes problems when, for example, subroutines are passed arguments which may be part of packed (and thus unaligned) data structures. This restriction was removed in the System/370 models, though one is warned that using unaligned data slows execution. A similar progression is made between the PDP-11 and VAX-11. Any data can occur on any address (almost), but, when a word or longword (or either half of a quadword) crosses a longword boundary, a micro-routine is invoked to do extra references.

1.2.3. Central Data Paths

The heart of the CPU is the thirty-two bit ALU and its associated registers. A simplified data path diagram is given in Figure 8, and a more complete diagram in Figure 9. The general registers (r0 - r14, but not PC) are kept in a duplicate pair or register files (RAB). This duplication provides for the contents of two different registers to be used at once. One set of registers' output passes through latch LA to the A (right) side of the ALU; the other passes through latch LB to its B (left) side. A file of temporaries (t0 - t7, and others) is also available on the ALU's left, after passing through latch LC. A set of sixty-four, sixteen-bit constants is also available on the ALU's left. Two very important registers, D and Q can be gated to either side of the
ALU - they have several special properties, especially in regards to shifting (to be discussed). The D register acts as the memory data register - all data routed to or from memory must pass through it. The "internal data" or ID bus takes its data from D and delivers to Q. This bus connects to several control registers (such as the alternate stack pointers, SCBB, page table origin and length registers), as well as the I-box - immediate and literal data is received by this path, as are branch displacements. Such data from the I-box come sign extended.

For testing the value of single bits, or constructing multi-bit masks for field insertion and extraction, a mask generator is available on the left ALU input. This circuit will yield 1's in all positions save one, where it will give a zero. That position is the one selected by the contents of the SC register (bit zero on the right). For example, to generate ones in word field <m:n>, and zeros elsewhere, takes five steps:

- load SC with m+1
- using the ALU operation A+B+1, and an A input of zero, store (mask+1) in a temporary, t0:
  
  \[
  \begin{align*}
  \text{mask} & : 1 \ldots 11011 \ldots 1 \\
  + & \hspace{1cm} 1 \\
  \text{t0} : & \hspace{1cm} 1 \ldots 11100 \ldots 0
  \end{align*}
  \]

- load SC with n
- as outlined above, store (mask+1) in the Q register:

\[
\begin{align*}
\text{Q} & : 1 \ldots 11110 \ldots 0 \\
\text{m} & \hspace{1cm} n \\
00 \ldots 0111 \ldots 10 \ldots 0
\end{align*}
\]

- perform the operation "Q ANDNOT t0". The result is:

\[
\begin{align*}
\text{m} \hspace{1cm} n \\
00 \ldots 0111 \ldots 10 \ldots 0
\end{align*}
\]

There are at least three ways in which shifting can be done in the central data paths. Two of these are for very specialized applications, the third is a general rotate unit. Between the ALU output and the D, Q, and file registers is the SHF box. Data passing through this can be shifted one or two
Figure 8 - Simplified CPU Data Path
places right, and one, two, or three places left, using various quantities as the shifted-in bits. The right-by-two shift is used in multiplication, where the product is formed, two bits at a time, in the Q and D registers. The left shift amount can be made to depend on the size of the operand currently under evaluation: zero for byte, one for word, two for longword, or three for quadword. In this way, the index operand of an indexed address specifier may be correctly scaled. In this case, zeros are always shifted in. The contents of the D or Q registers may also be shifted, left or right, one or two bits. Again, a double right shift is used in multiplication.

Finally, there is the full rotation unit. This takes the sixty-four bits from the Q and D registers (Q on the left), rotates by the amount specified by the contents of SC (or another source), and deposits thirty-two bits of the result back in D. A positive count denotes left rotation a negative count - right rotation.

The memory address register is called VA, for "virtual address". It and IBA can be loaded from the ALU output; which of them is used as a memory reference address depends on the destination of the data - VA is used for data fetches (via register D) and IBA for program stream fetches (to the I-box). Either of these registers may be loaded into the PC (which appears to the macro-programmer as register r15). The latter may also be incremented by one, two, four or n (a quantity determined by the instruction decode logic), using a dedicated adder - and avoiding use of the main ALU. Note that the VA and IBA registers cannot be gated directly through the ALU, but must pass through PC. Thus, whenever a microprogram interrupt is caused by a TB miss, and we desire to know the requested address causing the interrupt, we must save the PC (in an RC register), read VA through PC, then restore PC
through VA or IBA. Note also that PC cannot be loaded directly from ALU fan-out, but must pass through VA or IA. This presents less of a hardship, since loading PC usually indicates a program jump, and IBA must in that case be reloaded anyway.

In order to speed the handling of floating point quantities in machines without the optional floating point accelerator, an auxiliary, ten-bit ALU is provided. The major component in its data paths is the SC register, which is also for shifting and mask operations (as we have seen). Other registers associated with it are FE and STATE; constants can also be used. The STATE register (of eight bits) is often used to keep state information during the interpretation of complex instruction. For example, the subscript range check flag is kept here during the index instruction, since the tests are done early on, but no action should be taken until after the final result is stored. The decimal instructions also keep flags in this register. On the output of the exponent ALU (EALU) is a 256x8-bit ROM for looking up the negative absolute value of quantities. This is handy when it is necessary to de-normalize the smaller of two floating-point numbers by the difference of the exponents, for addition.

1.2.4. the Accelerator

The floating point accelerator is an optional piece of hardware designed to speed VAX floating point calculations. The accelerator is not merely a bus device, but one whose tentacles reach to may parts of the CPU. It recognizes pertinent opcodes in IB byte zero, and can affect the destination address of decision point (specifier decode) branches. It receives data by way of the ID bus, and returns results via a bus leading into the Q and D registers. It keeps its own copies of the general registers, and can set the condition code bits.
The sequence of events for a dyadic operation would go like this:

- First operand fetched into D register.
- D register gated to ID bus - accelerator signaled to receive data.
- Second operand fetched into D register.
- D register gated to ID bus - accelerator signaled to receive data.
- Microcode loops until accelerator signals result ready.
- Result gated into D or Q register - microcode acknowledges receipt of data.
- Accelerator condition code used to set PSL condition code bits, and operation result stored. Fault taken if V-bit (overflow) set.

1.2.5. Control Store and the Microsequencer

This massive mound of machinery, the CPU, is not controlled by random logic, but by a very large (about 57100 byte) micro-program. Control words are 96 bits wide, and divided into thirty fields (see Figure 10). The first 4096 words of control store are of ROM (and are known as PCS), the next 1024 are writeable (WCS) and used by the instruction set(s), by micro-ECO's, and by diagnostic routines. A further 1024 words of WCS are optional, and may be programmed by the user. Because of the great control word width, considerable parallelism is possible, but not often achieved, as most computations tend to be ALU and register bound. Thirteen bits of each microinstruction are used to form the address of the successor instruction. When straight-line microcode is being executed, this address is used directly. But, when any conditional branches are taken (governed by the BEN micro-word field), other information is also used. When BEN is non-zero, it selects one of twenty-six groups of three, four, or five condition bits. For example, group "1A" is the PSL condition code bits: N, Z, V, & C. These conditions, as they are queried at the beginning of a cycle, are ORed with the low-order bits of the address (the micro-word's J or JMP field) to form the address of the successor microword. As a further example, branch function "C" is:
<table>
<thead>
<tr>
<th>EALU</th>
<th>JMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>13 12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IEK</th>
<th>MSC</th>
<th>V</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30 29</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>MCT/CID</td>
</tr>
<tr>
<td>47</td>
<td>46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KMX</th>
<th>SI/ACM</th>
<th>QK</th>
<th>SGN</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
<td>55</td>
<td>54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DT</th>
<th>X</th>
<th>BEN</th>
<th>ACF</th>
<th>ALU</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>78</td>
<td>77</td>
<td>76</td>
<td>72</td>
<td>71</td>
</tr>
<tr>
<td>66</td>
<td>65</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IBC</th>
<th>DK</th>
<th>SHF</th>
<th>BMX</th>
<th>AMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>92 91</td>
<td>88 87</td>
<td>85 84</td>
<td>82 81 80</td>
</tr>
</tbody>
</table>

Figure 10 - VAX microinstruction format
That is, the condition (SC ≠ 0), and the two low-order bits of the D register. If all these conditions are tested, yielding an eight-way branch, the designated successor address of the testing instruction must end in binary "000". If, on the other hand, one only wants to do a four-way branch not involving the test of the SC register, the successor address should end in "100", effectively masking out the SC condition. The other addresses involved would end with "101", "110", and "111".

If the SUB field is one, a micro-subroutine call is specified, and the address of the current microword is pushed onto a (sixteen deep) stack before the branch is taken. If the SUB field is a two, an address is popped off this stack, and is ORed with the instruction's JMP field, as well as any conditions specified, to form the next word address. Note that if a word specifies a return, a zero address field, and no conditions, the returned-to address is exactly the one from which the call was made, and so the subroutine is recalled! A SUB field of value three denotes a decision point branch, and the lower eight bits of address are taken from the instruction decode logic.

The micro-code sequencer can be affected by several other machine conditions, such as a translation buffer miss.
2. MICROPROGRAMMING THE VAX

In the previous section, we saw some of the macro-architecture features of the VAX, and some of the micro-architecture supporting them. Here, we will investigate the micro-architecture on a much more detailed basis, with an emphasis on user microprogramming. Fields and functions not used or usable to the writer of user micro-code will be treated sketchily. Programming examples will mostly use the notation of DEC's assembler - that is, "FIELDNAME/value", where "value" is either a hexadecimal number or a symbolic constant. Field-value assignments separated by commas co-exist in the same micro-word. This assembler also employs a suggestive macro-notation, which I will not use.

2.1. ARITHMETIC SECTION

The Central Data Paths' Arithmetic Section (DEC's name) consists of the thirty-two-bit ALU, its input multiplexors, register files and latches, the constant and mask generation mechanisms, and some miscellany.

2.1.1. ALU functions and condition codes

The VAX's heart, the thirty-two-bit ALU, is built of 74S181 (ALU) and 74S182 (look-ahead carry generator) chips. Its function is controlled by the four-bit "ALU" field of the microword. This field is not used directly as the chip's function selector, but is mapped into a subset, controlling other functions as well. The function codes are shown in Figure 11.

The RLOG stack is used to record changes made to the general registers in the course of operand evaluation. For example, the macro-instruction ADDLI3 -(sp), -(sp), x requires that the stack pointer (r14) be twice decremented by four in order to fetch the operands of the addition, whose result is
then stored at location "x". If "x" is in an absent page, the instruction is interrupted, then re-executed after the absent page is fetched into primary memory. In order that the correct operands be re-fetched, any register modifications must be undone. Towards this end, the 16x9-bit RLOG stack can be used to record the lower four bits of the KMX field, the target register, and whether and add or subtract was done. An associated register, PCSV, saves the PC's low-order eight bits at each macro-instruction's beginning, so the PC may be restored in the case of an interrupt, and the operation correctly restarted.

When an "instruction dependent" operation is specified, a ROM in the instruction decode logic provides the ALU function select bits.

There are two sets of condition codes which may be set depending on the ALU's output. The PSL condition codes are accessible to the macro-program, whereas the micro-branch condition code (UBCC) is used for local
decision-making in the interpretation of an instruction. These codes are
governed by the value of micro-word field CCK, as shown in Figure 12. The
options available for condition-code setting seem peculiar for the garden-
variety arithmetic operation, so I assume most of these are covered by
"instruction dependent". The PSL V bit signifies an arithmetic overflow, and
C a carry. N is set when a result is negative, and Z when it is zero. The bits
tested to determine these conditions depend on the data type being
operated on. (For example, the sign of a byte operand is bit seven.) The DT
control-word field determines the operand length - at least for integers.
Values are shown in Figure 13. Note that the PSL condition code bits may
also be set from those of the floating-point accelerator after a floating-point

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>do nothing (default)</td>
</tr>
<tr>
<td>1</td>
<td>LOAD.UBCC</td>
<td>load UBCC from ALU and EALU conditions</td>
</tr>
<tr>
<td>2</td>
<td>SET.V</td>
<td>force PSL V bit</td>
</tr>
<tr>
<td>3</td>
<td>TST.Z</td>
<td>clear PSL Z if ALU ≠ 0</td>
</tr>
<tr>
<td>4</td>
<td>ROR</td>
<td>set PSL N &amp; Z from ALU, C from AMX&lt;0&gt;</td>
</tr>
<tr>
<td>5</td>
<td>N+Z_ALU</td>
<td>set PSL N &amp; Z from ALU</td>
</tr>
<tr>
<td>6</td>
<td>C_AMX0</td>
<td>set PSL C from AMX&lt;0&gt;</td>
</tr>
<tr>
<td>7</td>
<td>INST.DEP</td>
<td>instruction dependent</td>
</tr>
</tbody>
</table>

Figure 12 - CCK

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LONG</td>
<td>32-bit longword</td>
</tr>
<tr>
<td>1</td>
<td>WORD</td>
<td>16-bit word</td>
</tr>
<tr>
<td>2</td>
<td>BYTE</td>
<td>8-bit byte</td>
</tr>
<tr>
<td>3</td>
<td>INST.DEP</td>
<td>instruction dependent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(any of above, or quadword)</td>
</tr>
</tbody>
</table>

Figure 13 - DT
operation, by setting field MSC/8.

2.1.2. ALU inputs

The input to the ALU is chosen by two multiplexors: AMX and BMX. AMX, which supplies the ALU's A input, can select from two different sources; BMX gives eight choices for the B source.

2.1.2.1. AMX

The source selected by AMX is determined by control-word field AMX, as shown in Figure 14. LA is one of the latches connected to the register file RAB (see section 2.1.3). RAMX is another multiplexor, which can choose either the D register, or the Q register. It is controlled by microword field RMX, see Figure 15. This field also controls RBMX - a multiplexor with the same sources, which supplies BMX. Note that, since field RMX controls both these devices, one can never do an operation of the D or Q register with

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LA</td>
<td>latch LA (default)</td>
</tr>
<tr>
<td>1</td>
<td>RAMX</td>
<td>mux RAMX (register D or Q)</td>
</tr>
<tr>
<td>2</td>
<td>RAMX.SXT</td>
<td>RAMX, sign extended</td>
</tr>
<tr>
<td>3</td>
<td>RAMX.OXT</td>
<td>RAMX, zero extended</td>
</tr>
</tbody>
</table>

Figure 14 - AMX

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
<th>AMX</th>
<th>BMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D</td>
<td>Q</td>
<td>D</td>
<td>Q</td>
</tr>
<tr>
<td>1</td>
<td>Q</td>
<td>D</td>
<td>Q</td>
<td>D</td>
</tr>
</tbody>
</table>

Figure 15 - RMX (RAMX and RBMX)
itself; but that operations between the two will work. Sign or zero extension may be performed on the data supplied by RAMX, dictated by the data type field, DT. Thus, when a byte type is specified, and AMX selects RAMX, sign extended, the sign bit in position seven is propagated through bits <31:8>. A sign-extended longword is the same as the unmodified data from RAMX, but a zero-extended longword is identically zero. Thus coding "ALU/4, RAMX/3, DT/0" will result in the generation in the ALU of a quantity one greater than that supplied to the B-input by the BMX.

2.1.2.2. BMX

The B multiplexor is controlled by field BMX, as in Figure 16. The mask source is, as was previously discussed, a circuit providing 1's in all bit positions save the one selected by SC<4:0>. LB is a latch whose source is the register file RAB. LC is a latch whose source is the RC file (temporaries). PC is the program counter, which, although it appears as r15 to the macro-level programmer, is physically a separate register. A BMX field value of one selects the LB latch, unless the register selected (by the source specified in the SPO field, I presume) is r15, in which case the PC is used.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MASK</td>
<td>mask generator (default)</td>
</tr>
<tr>
<td>1</td>
<td>PC.OR.LB</td>
<td>LB, unless designating r15, then PC</td>
</tr>
<tr>
<td>2</td>
<td>PACKED.FL</td>
<td>pack floating number</td>
</tr>
<tr>
<td>3</td>
<td>LB</td>
<td>latch LB</td>
</tr>
<tr>
<td>4</td>
<td>LC</td>
<td>latch LC</td>
</tr>
<tr>
<td>5</td>
<td>PC</td>
<td>register PC</td>
</tr>
<tr>
<td>6</td>
<td>KMX</td>
<td>constant (or SC) from KMX</td>
</tr>
<tr>
<td>7</td>
<td>RBMX</td>
<td>mux RBMX (register D or Q)</td>
</tr>
</tbody>
</table>

Figure 18 - BMX
The B-mux can be used to assemble a floating-point format number from diverse sources, by specifying field BMX/2. The packed floating-point format is:

\[
\begin{array}{c|c|c|c}
\text{D<23:8>} & \text{SD} & \text{EALU<7:0>} & \text{D<30:24>}
\end{array}
\]

The exponent is supplied by the seven low-order bits of the output of the exponent ALU, and the fraction by the D register. The sign is provided by bit register SD (see paragraph 2.1.5). Due to timing delays in routing the data, both the EALU and ALU must be performing logical operations (no carries) to insure that the data is available at the ALU's output when required. A word containing the PCSV register and the top of the RLOG stack can be selected with "BMX/0, MSC/7" ("READ RLOG"). The format of this is:

\[
\begin{array}{c|c|c}
0 & \text{RLOG} & \text{PCSV}
\end{array}
\]

Constants may be introduced into the ALU by way of the constant multiplexor, governed by field KMX. Certain values come from the FK multiplexor, and may be used with impunity. Others are derived from a ROM called SK. These may be used in arithmetic ALU operations (involving a possible carry or borrow) only if an extra micro-instruction is allowed for set-up (?). KMX values are shown in Figure 17.

Register SC is (obviously) not a constant, but is routed through FK and KMX anyway. "Specifier 1 constant" appears to be the length of the operand currently under evaluation, as supplied by the I-box. This is useful since auto-increment address mode is supposed always to increment by this value. The constant four is also available for auto-increment deferred addressing. KMX/6 gets a zero in normal VAX mode, but has a different meaning in compatibility mode (namely, the size of the second operand - specifier 2 con-
Register SC is (obviously) not a constant, but is routed through FK and KMX anyway. "Specifier 1 constant" appears to be the length of the operand currently under evaluation, as supplied by the I-box. This is useful since auto-increment address mode is supposed always to increment by this value. The constant four is also available for auto-increment deferred addressing. KMX/8 gets a zero in normal VAX mode, but has a different meaning in compatibility mode (namely, the size of the second operand - specifier 2 constant).
2.1.3. the General and Temporary Register sets

Next to the ALU, register access is probably this machine's narrowest bottleneck. Despite the three latches, LA, LB, and LC, only one register may be read or written at a time (with special-case exceptions). This is because one seven-bit micro-word field, SPO, is used to control all register activity. This is divided into several subfields, as shown here, and tabulated in Figure
<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>do nothing (default)</td>
</tr>
<tr>
<td>8</td>
<td>LOAD.LC.SC</td>
<td>load LC from address in SC</td>
</tr>
<tr>
<td>7</td>
<td>WRITE.RC.SC</td>
<td>write RC register addressed by SC</td>
</tr>
</tbody>
</table>

**Field SPO**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD.LAB</td>
<td>load LA, LB from R(ACN)</td>
</tr>
<tr>
<td>2</td>
<td>LOAD.LC</td>
<td>load LA from R(RN), hold LB</td>
</tr>
<tr>
<td>3</td>
<td>WRITE.RAB</td>
<td>write RA, RB(ACN)</td>
</tr>
</tbody>
</table>

**Field SPO.AC**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SP1.SP1</td>
<td>select RAB from SP1</td>
</tr>
<tr>
<td>1</td>
<td>SP2.SP2</td>
<td>select RAB from SP2 (for r-r-op optimization)</td>
</tr>
<tr>
<td>3</td>
<td>PRN</td>
<td>select RAB from PRN</td>
</tr>
<tr>
<td>4</td>
<td>PRN+1</td>
<td>select register addressed by SC</td>
</tr>
<tr>
<td>5</td>
<td>SC</td>
<td>selected RAB from SP1+1</td>
</tr>
</tbody>
</table>

**Field SPO.ACN**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LOAD.LC</td>
<td>load LC from RC(RN)</td>
</tr>
<tr>
<td>3</td>
<td>WRITE.RC</td>
<td>write RC(RN)</td>
</tr>
<tr>
<td>4</td>
<td>LOAD.LAB</td>
<td>load LA, LB from RAB(RN)</td>
</tr>
<tr>
<td>5</td>
<td>WRITE.RAB</td>
<td>write RA, RB(RN)</td>
</tr>
<tr>
<td>6</td>
<td>LOAD.LAB.WRTE.RC</td>
<td>load LA, LB from R1, write RC(RN)</td>
</tr>
<tr>
<td>7</td>
<td>LOAD.LC.WRTE.RAB</td>
<td>write R1, load LC from RC(RN)</td>
</tr>
</tbody>
</table>

**Field SPO.R**

<table>
<thead>
<tr>
<th>VALUE</th>
<th>AS SPO.RAB symbol</th>
<th>register</th>
<th>AS SPO.RC symbol</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R0</td>
<td>r0</td>
<td>T0</td>
<td>t0</td>
</tr>
<tr>
<td>1</td>
<td>R1</td>
<td>r1</td>
<td>T1</td>
<td>t1</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>r2</td>
<td>T2</td>
<td>t2</td>
</tr>
<tr>
<td>3</td>
<td>R3</td>
<td>r3</td>
<td>T3</td>
<td>t3</td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
<td>r4</td>
<td>T4</td>
<td>t4</td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
<td>r5</td>
<td>T5</td>
<td>t5</td>
</tr>
<tr>
<td>6</td>
<td>R6</td>
<td>r6</td>
<td>T6</td>
<td>t6</td>
</tr>
<tr>
<td>7</td>
<td>R7</td>
<td>r7</td>
<td>T7</td>
<td>t7</td>
</tr>
<tr>
<td>8</td>
<td>r8</td>
<td>LC.SV</td>
<td>t8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r9</td>
<td>VA.SV</td>
<td>t9</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>r10</td>
<td>PTE.VA</td>
<td>t10</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>r11</td>
<td>PTE.PA</td>
<td>t11</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>AP</td>
<td>r12</td>
<td>PC.SV</td>
<td>t12</td>
</tr>
<tr>
<td>D</td>
<td>FP</td>
<td>r13</td>
<td>SC.SV</td>
<td>t14</td>
</tr>
<tr>
<td>E</td>
<td>SP</td>
<td>r14</td>
<td>VA.REF</td>
<td>t15</td>
</tr>
<tr>
<td>F</td>
<td>R15</td>
<td>r15</td>
<td>MBT.VA</td>
<td>t15</td>
</tr>
</tbody>
</table>

**Field SPO.RAB & SPO.RC**

**Figure 18 - Subdivisions of SPO**
LA, LB, and LC are latches, so data may be clocked out of a register and used in a logical ALU operation in the same micro-word. Because of timing constraints, however, an extra cycle should be allowed for operations involving carry propagation. The latches' contents will remain unchanged, and may be used in later operations, as long as no further register contents have been read through them. You should not count on their being unchanged at any time when the microprogram may be interrupted (as when doing memory accesses). SPO field values are more rationally explicated in Figure 19, where: SP1 signifies the register designated by the operand specifier currently under evaluation; SP2 is the register number of the next specifier (useful, perhaps, in optimizing register-register instructions); and PRN (previous register number) is the register of the last specifier evaluated. In PDP-11 mode, SRC and DEST are the two operand registers of the instruction word.

The RC registers always act as full, longword registers. However, when loading an RAB register, a partial word may be loaded, depending on the data type, determined by field DT. The SPO values useful to the microprogrammer are of the form 2x (read an RC), 3x (write RC), 4x (read RAB), and 5x (write RAB). The address modes allowing one to use the SC register as an index are also useful, and facilitate multiple register loading and storing, as for a context switch or subroutine entry (where the registers to be saved are
designated by 1's in a bit map). The other register specifying schemes are of limited application.

2.1.4. SHF

A limited shifter, SHF, takes the ALU output, and feeds the register files, the D register, the Q register, and the accelerator. It is mainly used in multiplication, division, and subscript scaling, as has been mentioned. It is controlled by microword field SHF, as shown in Figure 20. "Data dependent" shift amounts are determined by the DT field, and the decode logic. For SHF

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ALU</td>
<td>no shift (default)</td>
</tr>
<tr>
<td>1</td>
<td>LEFT</td>
<td>shift left one</td>
</tr>
<tr>
<td>2</td>
<td>RIGHT</td>
<td>shift right one</td>
</tr>
<tr>
<td>3</td>
<td>ALU.DT</td>
<td>shift left by data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(byte 0, word 1, long 2, quad 3)</td>
</tr>
<tr>
<td>4</td>
<td>RIGHT2</td>
<td>shift right 2</td>
</tr>
<tr>
<td>5</td>
<td>LEFT3</td>
<td>shift left 3</td>
</tr>
</tbody>
</table>

Figure 20 - SHF
field values of three or five, the quantity shifted in is always zero. For the other cases, it depends on the value of field SI. These dependencies are shown in Figure 21.

2.1.5. Sign Control

When performing floating-point calculations in the Central Data Paths, the signs of the operands are kept in two flip-flops: SS and SD. They can be loaded from bit fifteen of ALU output (the sign bit of an assembled floating-point number), and a small number of operations may be performed on them. They are controlled by field SGN, as in Figure 22. Note the operation "SS <- SS xor ALU<15> xor IR<1>." Floating add instructions have a zero in opcode bit 1, and floating subtracts a one. Thus similar instructions may use common microcode (as in the case of "instruction dependent" operations).

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DIVD</td>
<td>PSL&lt;N&gt; ALU C&lt;31&gt;</td>
</tr>
<tr>
<td>1</td>
<td>ASHR</td>
<td>ALU&lt;31&gt; Q&lt;31&gt;</td>
</tr>
<tr>
<td>2</td>
<td>ASHL</td>
<td>0 D&lt;31&gt;</td>
</tr>
<tr>
<td>3</td>
<td>ZERO</td>
<td>0 0</td>
</tr>
<tr>
<td>5</td>
<td>DIV</td>
<td>Q&lt;31&gt; ALU C&lt;31&gt;</td>
</tr>
<tr>
<td>6</td>
<td>MUL+</td>
<td>0 0 ALU&lt;1:0&gt;</td>
</tr>
<tr>
<td>7</td>
<td>MUL-</td>
<td>0 0 ALU&lt;1:0&gt;</td>
</tr>
</tbody>
</table>

Figure 21 - SI

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>do nothing (default)</td>
</tr>
<tr>
<td>1</td>
<td>LOAD.SS</td>
<td>SS</td>
</tr>
<tr>
<td>2</td>
<td>SS.FROM.SD</td>
<td>SS -&gt; SD</td>
</tr>
<tr>
<td>3</td>
<td>NOT.SD</td>
<td>SD -&gt; SD</td>
</tr>
<tr>
<td>4</td>
<td>SD.FROM.SS</td>
<td>SD -&gt; SS</td>
</tr>
<tr>
<td>5</td>
<td>SS.XOR.ALU</td>
<td>SD -&gt; ALU&lt;15&gt;, SS -&gt; SS ⊕ ALU&lt;15&gt;</td>
</tr>
<tr>
<td>6</td>
<td>ADD.SUB</td>
<td>SD -&gt; ALU&lt;15&gt;, SS -&gt; SS ⊕ ALU&lt;15&gt; ⊕ IR&lt;1&gt;</td>
</tr>
<tr>
<td>7</td>
<td>CLR.SD+SS</td>
<td>clear SD and SS</td>
</tr>
</tbody>
</table>

Figure 22 - SGN

SC is the source of the sign when floating-point numbers are re-assembled in the ALU's B-mux.
2.2. DATA SECTION

The Data section is that part of the Central Data Paths which includes the D and Q register, the shifter, the interfaces to the ID bus, accelerator, and memory data.

2.2.1. Data Format Multiplexor

Data from the ALU can be gated into the D and Q registers. Its path from the ALU is through shifter SHF, and through a curious device, the data formatter multiplexor (DFMX), on its way to the D and Q registers' input multiplexors. One may specify (by way of fields QK and DK) that data be transmitted from the ALU in either integer or unpacked floating-point format. In integer mode, the thirty-two-bit word is received just as it leaves SHF. Or, data assumed to be the floating point format previously exhibited may be unpacked by DFMX into the following format:

\[
\begin{align*}
&01 &\text{SHF}<6:0> &\quad &22 &\text{SHF}<3i:16> &\quad &6 &\quad &0 \\
\end{align*}
\]

Note that this is not exactly the converse to the packing operation of the B-mux, and that the leading 1 implicit in normalized floating-point numbers has been made explicit, in bit 30.

2.2.2. DAL

The general, sixty-four-bit shifter (Data Aligner, DAL) was discussed in an earlier section. Actually, the shifting scheme used is more ingenious than was presented. There are three levels of shifting circuitry. Level one is governed by SC bits nine and four, and can shift left by 0, 16, 32 (same as a right shift of 32), or 48 (same as a right shift of 16). Level two is governed by
SC bits <3:2>, and shifts left by 0, 4, 8, or 12 bits. The third level, governed by SC<1:0>, shifts left 0, 1, 2, or 3. So, if SC contained "1xxxx1111", the rotation would be (right 16)+(left 12)+(left 3) = right-by-one.

2.2.3. Accelerator Data

Output data from the floating point accelerator is available to the Q and D registers on the same bus fed by DFMX, and may be gated into them when the accelerator signals it is ready. This device keeps its own copies of the general registers, and loads a register with the data from this bus whenever the corresponding register in the set RAB is loaded. This implies that whenever a register is loaded by the microprogram, DFMX had better be selected for integer-format data, or the accelerator's register will be incorrectly loaded!

2.2.4. ID Bus

The registers on the internal data (ID) bus may be inspected and loaded by way of the Q and D registers. These functions are controlled by the CID, FS, and KMX (alias ID.ADDR) microword fields. Coding "FS/1, CID/5" causes the ID register selected by the KMX field (not the constant value selected) to be read - it can thus be gated into the Q register; "FS/1, CID/4" causes the ID bus register selected by the contents of register SC to be read. "FS/1, CID/7" writes the contents of the D register in the ID register selected by field KMX; "FS/1, CID/6" causes the bus register modified to be chosen by the contents of register SC. The ID bus registers' addresses are shown in Figure 24. Many of these registers are of interest only for diagnostic purposes - these registers are available to the LSI-11, too, so can be read and diagnosed by a console program. Other registers are visible to the macro-programmer.
Figure 23 - ID bus Control

<table>
<thead>
<tr>
<th>register address source</th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>field KMX</td>
<td>FS/1, CID/5</td>
<td>FS/1, CID/7</td>
</tr>
<tr>
<td>register SC</td>
<td>FS/1, CID/4</td>
<td>FS/1, CID/6</td>
</tr>
</tbody>
</table>

Figure 24 - ID Bus registers

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IBUF</td>
<td>data from IB</td>
</tr>
<tr>
<td>1</td>
<td>DAY.TIME</td>
<td>current time of day</td>
</tr>
<tr>
<td>3</td>
<td>SYS.ID</td>
<td>system id register</td>
</tr>
<tr>
<td>4</td>
<td>RXCS</td>
<td>Console receive status register</td>
</tr>
<tr>
<td>5</td>
<td>RXDB</td>
<td>Console receive data byte</td>
</tr>
<tr>
<td>6</td>
<td>TXCS</td>
<td>Console transmit status register</td>
</tr>
<tr>
<td>7</td>
<td>TXDB</td>
<td>Console transmit data byte</td>
</tr>
<tr>
<td>8</td>
<td>DQ</td>
<td>D/Q registers (maintenance use)</td>
</tr>
<tr>
<td>9</td>
<td>NXT.PER</td>
<td>Interval Clock next period</td>
</tr>
<tr>
<td>A</td>
<td>CLK.CS</td>
<td>Interval Clock control</td>
</tr>
<tr>
<td>B</td>
<td>INTERVAL</td>
<td>Current interval count</td>
</tr>
<tr>
<td>C</td>
<td>CES</td>
<td>CPU error/status</td>
</tr>
<tr>
<td>D</td>
<td>VECTOR</td>
<td>Exception control</td>
</tr>
<tr>
<td>E</td>
<td>SIR</td>
<td>Software Interrupt Register</td>
</tr>
<tr>
<td>F</td>
<td>PSL</td>
<td>Processor Status Longword</td>
</tr>
<tr>
<td>11</td>
<td>TBUF</td>
<td>Translation buffer data</td>
</tr>
<tr>
<td>12</td>
<td>TBERO</td>
<td>TB error/stat 0</td>
</tr>
<tr>
<td>13</td>
<td>TBER1</td>
<td>TB error/stat 1</td>
</tr>
<tr>
<td>14</td>
<td>ACC.0</td>
<td>Accelerator register 0</td>
</tr>
<tr>
<td>15</td>
<td>ACC.1</td>
<td>Accelerator register 1</td>
</tr>
<tr>
<td>16</td>
<td>ACC.2</td>
<td>Accelerator register 2</td>
</tr>
<tr>
<td>17</td>
<td>ACC.CS</td>
<td>Accelerator control/stat</td>
</tr>
<tr>
<td>18</td>
<td>SILO</td>
<td>Next item of SBI history</td>
</tr>
<tr>
<td>19</td>
<td>SBI.ERR</td>
<td>SBI error register</td>
</tr>
<tr>
<td>1A</td>
<td>TIME.ADDR</td>
<td>SBI timeout address</td>
</tr>
<tr>
<td>1B</td>
<td>FAULT</td>
<td>fault/status</td>
</tr>
<tr>
<td>1C</td>
<td>COMP</td>
<td>SBI silo comparator</td>
</tr>
<tr>
<td>1D</td>
<td>MAINT</td>
<td>SBI maintenance</td>
</tr>
<tr>
<td>1E</td>
<td>PARTY</td>
<td>Cache parity</td>
</tr>
<tr>
<td>20</td>
<td>USTACK</td>
<td>Microstack</td>
</tr>
<tr>
<td>21</td>
<td>UBREAK</td>
<td>Micro-break address</td>
</tr>
<tr>
<td>22</td>
<td>WCS.ADDR</td>
<td>WCS write address</td>
</tr>
<tr>
<td>23</td>
<td>WCS.DATA</td>
<td>WCS write data</td>
</tr>
</tbody>
</table>

and are adequately described in the VAX-11/780 handbook series.

The data from the instruction buffer is usually a byte, word, or longword of immediate data. When IBC/7 is coded in the same micro-word that causes
this source to be read, the data received is a sign-extended byte (or word) to
be used as a branch displacement.

ID registers 30 - 39 are also named T0 - T9. These are temporary registers used primarily during a machine error logout (to store the logout data, in case writing to memory is infeasible). They are otherwise free to the micro-programmer.

2.2.5. the Q Register

The Q register serves as a source of data for shifter and ALU operations, as described previously. It may be loaded from several sources, including DFMX, the ID bus, and the accelerator. This is controlled by microword field QK, as shown in Figure 25. Those features not previously mentioned are its shifting ability, and a decimal correction factor.

The contents of the Q register may be shifted left or right by one or two bits. The bits shifted in depend on the setting of field SI, as shown in Figure 21.

The VAX instruction set supports packed (BCD) decimal arithmetic instructions. However, the VAX ALU only operates on binary quantities.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>hold value (default)</td>
</tr>
<tr>
<td>1</td>
<td>LEFT2</td>
<td>shift left 2</td>
</tr>
<tr>
<td>2</td>
<td>RIGHT2</td>
<td>shift right 2</td>
</tr>
<tr>
<td>5</td>
<td>LEFT</td>
<td>shift left 1</td>
</tr>
<tr>
<td>6</td>
<td>RIGHT</td>
<td>shift right 1</td>
</tr>
<tr>
<td>8</td>
<td>SHF</td>
<td>load from SHF, integer format</td>
</tr>
<tr>
<td>9</td>
<td>SHF.FL</td>
<td>load from SHF, unpacked floating format</td>
</tr>
<tr>
<td>A</td>
<td>DEC.CON</td>
<td>load decimal correction factor</td>
</tr>
<tr>
<td>B</td>
<td>ACCEL</td>
<td>load accelerator data</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>load from register D (via DAL)</td>
</tr>
<tr>
<td>E</td>
<td>ID</td>
<td>load ID bus data</td>
</tr>
<tr>
<td>F</td>
<td>CLR</td>
<td>clear register Q</td>
</tr>
</tbody>
</table>

Figure 25 - QK
Thus, it is sometimes necessary to add 6's into a four-bit decimal digit (nibble) to force a carry. The usual algorithm for adding together word-fulls of decimal digits, A and B, is:

1. \( T \leftarrow A + \) (a word containing "0110" in each decimal digit)
2. \( B \leftarrow B + T \)
3. \( T \leftarrow \) (a word containing "0110" in each decimal digit for which a carry-out was not generated in the previous step, else zero)
4. \( B \leftarrow B - T \)

Steps two and three of this procedure are usually done in parallel, as the nibble-carry information does not persist.

A mechanism to aid this computation is the "decimal constant" which may be loaded into the Q register. (I don’t know how this works, but my intuition is that it works the same as a similar mechanism of the QM-1 computer: it will generate the all-6's word of step one when zero is passed through the ALU, and the 0's-and-6's word of step three when the partial sum of step two is generated in the ALU.)

2.2.6. the D Register

The D register is a source of data for shifter and ALU operations, memory and ID bus transactions. It may be loaded from several sources, including DFMX, memory data, DAL, and the accelerator. This is controlled by microword field DK, as shown in Figure 26.

The contents of the D register may be shifted left or right by one or two bits. The bits shifted in depend on the setting of field SI, for which see Figure 21. This register can be conditionally shifted left by one place; if a carry-out of the ALU's most significant bit (31) is generated, the shift is done (shifted-in bit depending on SI), otherwise, the register is loaded with data from SHF.
Packed decimal strings are held in memory with the highest-order byte at the lowest address. So the BCD representation of the number 1234567 in a longword would be

\[ 7 \times 563412 \]

since, in a longword, the low-order byte has the lowest address. Computationally, it would be convenient if the most significant digit were on the left, and the least significant on the right. The byte swap mechanism can transform the above representation, once loaded into the D register, into

\[ 1234567+ \]

the desired form, also in register D.

The D register serves as the memory data register for both reads and writes. Memory reads are always done in longwords - but a program can address data on byte boundaries. Between the memory data (MD) bus and the D register, the memory data aligned (MDAL) rotates the incoming

\[ x \] denoting the BCD representation of \( x \).
longword from the cache according to the two low-order bits of the requested address (from VA). In the case where data crosses a longword boundary (as determined by the address, and the size specified by the DT field) a second read must be done. The incoming data from this second read is prevented from clobbering the usable data already in the D register by a validity bit mechanism (similar to that of the instruction buffer). A complementary mechanism governs the insertion of partial words into memory, during store operations. Note that partial-register data from memory should be sign or zero extended appropriately before use.

2.3. ADDRESS SECTION

The address section is that part of the Central Data Paths concerned with the generation of addresses for accessing main store. This includes registers VA, IBA, and PC. Generally, when fetching program stream to the instruction buffer, IBA is the address source, and when accessing data, VA is the address source. (This statement is an oversimplification, and will be dealt with presently.)

2.3.1. Register VA

Register VA is used as the source of virtual addresses for a data fetch or store operation. In this case, the address is translated before use by the translation buffer mechanism. Its contents can also be interpreted as a physical address, as, for example, when accessing the system control block or system page table (which are addressed in physical memory). Or, it can be used as an index into the translation buffer, without any data transmission taking place, as in the case of a PROBE instruction. This is also its use when invalidating or manipulating TB entries (a PTE's modify bit is updated by a
microcoded routine when the corresponding page is first modified - TB entries are accessed via the ID bus). And, when a program jump is taken, the destination should be loaded in VA as well as IBA. VA is loaded from ALU fan-out when field VAK is set to one.

2.3.2. IBA

The IBA (or more properly, VIBA, for virtual instruction-buffer address) holds the virtual address where the IB is fetching instruction stream bytes. It is loaded from ALU output by setting micro-instruction field IBC to two, and is automatically updated by the instruction buffer as successive long-

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>hold value (default)</td>
</tr>
<tr>
<td>1</td>
<td>LOAD</td>
<td>load VA from ALU output</td>
</tr>
</tbody>
</table>

a) field VAK

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>no control (default)</td>
</tr>
<tr>
<td>1</td>
<td>STOP</td>
<td>flush IB, load IBA from ALU output</td>
</tr>
<tr>
<td>2</td>
<td>FLUSH</td>
<td>clear bytes 0 &amp; 1 (PDP-11 instruction)</td>
</tr>
<tr>
<td>3</td>
<td>START</td>
<td>clear bytes 2 &amp; 3 (PDP-11 data)</td>
</tr>
<tr>
<td>4</td>
<td>CLR.0.1</td>
<td>transfer branch displacement on ID bus</td>
</tr>
<tr>
<td>5</td>
<td>CLR.2.3</td>
<td>clear byte 0 (VAX opcode)</td>
</tr>
<tr>
<td>7</td>
<td>BDEST</td>
<td>clear byte 1 (VAX specifier)</td>
</tr>
<tr>
<td>C</td>
<td>CLR.0</td>
<td>clear bytes 0 - 3</td>
</tr>
<tr>
<td>D</td>
<td>CLR.1</td>
<td>clear bytes 1 - 5 conditionally.</td>
</tr>
<tr>
<td>E</td>
<td>CLR.0-3</td>
<td>If there is no specifier evaluation, clear nothing.</td>
</tr>
<tr>
<td>F</td>
<td>CLR.1-5.COND</td>
<td>If a self-contained specifier, clear it.</td>
</tr>
</tbody>
</table>

b) field IBC

Figure 27 - fields VAK and IBC
words are fetched. The low-order two bits determine how data is byte-rotated as it enters the IB, and are adjusted as the IB justifies its data requests to longword boundaries. This register is used as the actual address for a memory fetch only when there is an interruption to sequential program flow (as in the cases of a program jump, successful branch, interrupt or exception, or the crossing of a page boundary). On these occasions, a physical IBA register (IPA) is set up by the TB mechanism, and used thereafter. This ingenious design avoids the translation process for most program fetches. IBA is loaded from ALU fan-out when the IB is flushed by coding IBC/2.

2.3.3. PC

At the beginning of the interpretation of each macro-instruction, the program counter (PC) register contains the address of the opcode byte. When interpreting an operand, the PC addresses the operand specifier, and can be used in computing relative addresses. The upper twenty-eight bits of PC comprise a counter, and the lower bits can be loaded from the output of a four-bit ALU - the carry-out of which is used to increment the twenty-eight-bit counter. In this way, small constants (1, 2, 4, or \( n \), a number determined by the I-box) may be added to PC, see Figure 28. Since this register is often updated, incrementing it via the thirty-two-bit ALU would cause instruction interpretation to be considerably more ALU-bound than by the present scheme. Since PC is often used by seldom loaded, it should have a direct path to an ALU input, but does not need to be easily loaded from ALU output. In fact, loading PC usually signifies a change from sequential program control flow, so IBA and VA must be loaded with the same address, and PC may be loaded from one of these. PC operations are controlled by micro-instruction
field PCK, as shown in Figure 29.

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>do nothing (default)</td>
</tr>
<tr>
<td>1</td>
<td>PC_YA</td>
<td>PC ← VA</td>
</tr>
<tr>
<td>2</td>
<td>PC_JBA</td>
<td>PC ← IBA</td>
</tr>
<tr>
<td>3</td>
<td>VA+4</td>
<td>VA ← VA + 4</td>
</tr>
<tr>
<td>4</td>
<td>PC+1</td>
<td>PC ← PC + 1</td>
</tr>
<tr>
<td>5</td>
<td>PC+2</td>
<td>PC ← PC + 2</td>
</tr>
<tr>
<td>6</td>
<td>PC+4</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td>7</td>
<td>PC+N</td>
<td>PC ← PC + n</td>
</tr>
</tbody>
</table>

Figure 29 - PCK
2.4. EXPONENT SECTION

The Exponent Section is a part of the Central Data Paths designated for the handling of floating point exponents, in the absence of a floating-point accelerator. This includes three registers and a ten-bit ALU, so that a number’s exponent and fraction can be handled concurrently. Floating point numbers can be assembled and disassembled using the ALU B-mux and the DFMX, as we have already seen.

2.4.1. the Exponent ALU

The EALU takes its A input from either of registers SC or STATE, and its B input from a variety of sources; including register FE and the output from the constant multiplexor, KMX. Its output can be used to load any of the registers FE, SC, and STATE. The EALU A-mux (EAMX) is controlled by field MSC - when MSC/5 is coded, the STATE register is selected as a data source, otherwise register SC is used. The EALU B-mux (EBMX) is controlled by field EBMX, as shown in Figure 30. Note that, since the contents of the SC register can be routed through the constant multiplexor, they can be used as a source for the EALU’s B input. The EALU function is controlled by microword field EALU, see Figure 31. The negative-absolute-value function invokes a 256x8-bit ROM on the output of the EALU. A difference generated in the EALU

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FE</td>
<td>choose register FE</td>
</tr>
<tr>
<td>1</td>
<td>KMX</td>
<td>choose constant multiplexor</td>
</tr>
<tr>
<td>2</td>
<td>AMX.EXP</td>
<td>exponent part of AMX data</td>
</tr>
<tr>
<td>3</td>
<td>SHF.VAL</td>
<td>&quot;shift value&quot;??</td>
</tr>
</tbody>
</table>

Figure 30 - EBMX
is used as an index into the ROM, which supplies a negative shift count, as when one wants to denormalize the fraction part of a floating point number by the difference of two exponents. Which of two fractions is to be shifted depends on the true sign of the difference, so one hopes that the EALU sign can be tested before it is changed by the NABS mechanism.

2.4.2. Shift Count Register

The ten-bit SC register is loaded from one of several sources when field SCK is set to one, otherwise the contents remain unchanged. When SC is loaded, it may be from one of several sources; this function is controlled by field SMX, Figure 32. SC is a source for the A input to the EALU, and the B inputs of both the thirty-two bit ALU and the EALU, via KMX. Additionally, it governs the generation of bit masks, the shifting of data in the DAL unit, and can be used as an index to the scratch pad registers (RAB and RC) and the ID bus registers.

2.4.3. the FE register

FE is a ten-bit register used in floating-point exponent computations in the EALU. It is loaded from EALU output when field FEK is set to one, else the
### SCK

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>hold value (default)</td>
</tr>
<tr>
<td>1</td>
<td>LOAD</td>
<td>load SC from SMX</td>
</tr>
</tbody>
</table>

### SMX

<table>
<thead>
<tr>
<th>VAL</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EALU</td>
<td>EALU output (default)</td>
</tr>
<tr>
<td>1</td>
<td>FE</td>
<td>register FE</td>
</tr>
<tr>
<td>2</td>
<td>ALU</td>
<td>ALU&lt;9:0&gt;</td>
</tr>
<tr>
<td>3</td>
<td>ALU.EXP</td>
<td>ALU&lt;14:7&gt;</td>
</tr>
</tbody>
</table>

Figure 32 - SCK and SMX

...contents are retained. It is a source of data for the EALU's B-input, and can be loaded directly into SC.

#### 2.4.4. STATE

The eight-bit STATE register is used, as the name implies, to encode micro-program state information. Each of its four-bit halves may be used to...
control a multi-way microcode branch of up to sixteen different destinations. This register may be loaded with the output of the EALU when the MSC field is set to five. Note that this is the same field which governs the EALU's A-input - whenever STATE is selected as an EALU source, it is loaded from EALU output. This would seem to limit STATE's usefulness as an arithmetic register, but is certainly adequate for setting and clearing flags, which are, after all, the register's intended contents.

2.5. HOW TO USE MEMORY

There are several flavors of memory access. These are described in chapter eight of the VAX Hardware Handbook. For our purposes, there are four types of memory access - data read, data write, sequential program fetch, and non-sequential program fetch (at a new address). Memory is controlled primarily by control word field MCT, as shown in Figure 34.

2.5.1. Address Sources

The source of memory access addresses depends on whether the data is destined for the IB. If so, IBA is considered the address source, otherwise, VA is used. Only longword addresses are sent to memory - the lower two bits of a byte address determine the byte within the longword, and thus determine byte rotation at the destination, upon reading, and the generation of a mask, upon writing. The source of address bits is shown in Figure 35. Note that the low-order bits are always taken from VA. The implications of this are not what might be imagined, as we shall see.
<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TEST.RCHK</td>
<td>probe for readability</td>
</tr>
<tr>
<td>2</td>
<td>MEM.NOP</td>
<td>do nothing</td>
</tr>
<tr>
<td>4</td>
<td>TEST.WCHK</td>
<td>probe for writability</td>
</tr>
<tr>
<td>A</td>
<td>WRITE.V.NOCHK</td>
<td>write, no traps</td>
</tr>
<tr>
<td>C</td>
<td>WRITE.V.WCHK</td>
<td>normal write</td>
</tr>
<tr>
<td>E</td>
<td>LOCKWRITE.V.XCHK</td>
<td>interlock write</td>
</tr>
<tr>
<td>10</td>
<td>READ.V.RCHK</td>
<td>normal read</td>
</tr>
<tr>
<td>13</td>
<td>READ.V.NOCHK</td>
<td>read, no traps</td>
</tr>
<tr>
<td>14</td>
<td>READ.V.WCHK</td>
<td>read, for modify</td>
</tr>
<tr>
<td>16</td>
<td>READ.V.IBCHK</td>
<td>read, check controlled by Ibuf</td>
</tr>
<tr>
<td>18</td>
<td>READ.V.NEWPC</td>
<td>restart instruction fetching</td>
</tr>
<tr>
<td>1A</td>
<td>LOCKREAD.V.NOCHK</td>
<td>interlock read, inhibit check</td>
</tr>
<tr>
<td>1C</td>
<td>LOCKREAD.V.WCHK</td>
<td>interlock read</td>
</tr>
<tr>
<td>20</td>
<td>SBI.HOLD</td>
<td>stop SBI activity</td>
</tr>
<tr>
<td>22</td>
<td>SBI.HOLD+UNJAM</td>
<td>reset SBI</td>
</tr>
<tr>
<td>24</td>
<td>INVALIDATE</td>
<td>clear cache entry</td>
</tr>
<tr>
<td>26</td>
<td>VALIDATE</td>
<td>make cache entry valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(for microdiagnostics)</td>
</tr>
<tr>
<td>28</td>
<td>EXTWRITE.P</td>
<td>extended write to clear parity errors</td>
</tr>
<tr>
<td>2A</td>
<td>WRITE.P</td>
<td>physical write</td>
</tr>
<tr>
<td>2E</td>
<td>LOCKWRITE.P</td>
<td>physical interlock write</td>
</tr>
<tr>
<td>32</td>
<td>READ.P</td>
<td>physical read</td>
</tr>
<tr>
<td>36</td>
<td>READ.INT.SUM</td>
<td>&quot;interrupt summary read&quot;??</td>
</tr>
<tr>
<td>3A</td>
<td>LOCKREAD.P</td>
<td>physical interlock read</td>
</tr>
<tr>
<td>3E</td>
<td>ALLOW.IB.READ</td>
<td>let IB fetchahead (default)</td>
</tr>
</tbody>
</table>

Figure 34 - MCT

<table>
<thead>
<tr>
<th>VAX</th>
<th>VA&lt;31:9&gt;</th>
<th>VA&lt;8:2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIBA</td>
<td>VIBA&lt;31:9&gt;</td>
<td>VA&lt;8:2&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDP-11</th>
<th>VA&lt;15:9&gt;</th>
<th>VA&lt;8:2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>VA&lt;15:9&gt;</td>
<td>VA&lt;8:2&gt;</td>
</tr>
<tr>
<td>VIBA</td>
<td>VA&lt;15:9&gt;</td>
<td>VA&lt;8:2&gt;</td>
</tr>
</tbody>
</table>

Figure 35 - Address Bits Source

2.5.2. Sequential Program Fetching

When you aren't doing anything else with memory, you may as well let IB fetch ahead, so that program stream bytes will be available when required; the default setting for field MCT is to allow this. Pre-fetching can be avoided in a variety of ways, the simplest of which is to code memory no-ops in each
micro-instruction. This action can also be turned off by stopping IBA, using
field IBC. (Why you would want to do this I do not know.)

Recall that the address source for sequential program fetching is really
IPA, a physical address, rather than IBA, a virtual address. IPA is updated by
IB at the same time as IBA, but whenever the low order bits of IPA (which
needs only to contain longword addresses) become zero, a page boundary
has been crossed, and the high-order twenty-three bits of IBA must be re-
translated, to yield the new page frame number. So, IBA<8:0> is never
required for a sequential program fetch.

2.5.3. Non-Sequential Program Fetching

Whenever a program control jump occurs - either through a branch
instruction or an interrupt or exception, several things have to be done:
erroneously fetched-ahead bytes in the IB must be flushed; IBA and PC must
be set anew, as must IPC; IB fetching must be restarted. Because the entire
destination address must be translated in order to load IPC, and because VA
is always the source of low-order address bits, VA must also be loaded with
the destination address. Since you probably weren't going to do anything
further with the old contents of VA anyway, and since VA and IBA can be
loaded in parallel from ALU output, this won't cost you any time (and saves a
few bits of mux). A program jump is usually the last thing done in the
interpretation of an instruction. A formula for doing this is:

(generate destination address in ALU),
VAK/LOAD, PCK/PC_VA, ; note how VA is load-through
IBC/FLUSH, IEK/ISTR ; throw away old IB contents,
; see if interrupt pending
; ------------------------
PCK/PC+1, VAK/NOP, ; increment PC past new opcode
MCT/READ.V.NEWPC, ; restart IB fetches
J/62 ; go do next instruction (62 hex)
2.5.4. Data Reads

To read data to be used in computations, simply load VA with the virtual address of the datum (or physical address, for physical reads), and in a succeeding micro-instruction, code the read operation, including no-ops in fields VAK and DK, and an appropriate data type in DT. The data should be available in the D register in the next micro-instruction. However, the mechanism involved is much more complicated that this indicates. If the TB does not contain an entry for the virtual address requested, a micro-program interrupt gives control to a routine to fetch the appropriate PTE(s), then returns to the reading microroutine (with perhaps, some registers modified—so be careful). If the page is invalid (either missing from main memory, or a bad address, or no permission), then you lose control completely, and a macro-program interrupt occurs. If the requested address can be translated, but the data is not in the cache, the micro-program will be forced to execute no-op instructions until the data is available. (Note how this may change the effect of any timing-dependent operations strung between the reading micro-instruction and its successor.) Finally, if the memory address and data length coded in DT determine that the required datum does not reside in a single longword, a micro-program interrupt gives control to a routine to do the second read, to fetch the rest of this operand. Control is then returned to the reading routine.

2.5.5. Data Writes

Writing data is pretty much the same as reading it; put address in VA, put data in D, write. And just as many nasty things can happen to you—including a micro-interrupt to set the PTE's modify (dirty) bit, if this is the first modification of a page since that bit was cleared. Not to mention setting
the modify bit of the system PTE addressing your PTE.
2.6. MICROPROGRAM SEQUENCING

Just as in the case of macro-programs, a micro-program is a series of instructions to be executed, one after another, in some order specified by the programmer. The notions of conditional and unconditional sequencing, of subroutines, and even of interrupts, are shared by the macro- and micro-programmer. However, the mechanisms present in the VAX micro-architecture to implement these concepts are considerably different from those used in most macro-architectures, and for that reason need to be discussed. VAX micro-program addresses can come from a variety of sources, depending on conditions in the processor. These conditions are, from highest to lowest priority:

- initialize
- maintenance return
- cache stall
- micro-trap
- micro-ECO
- normal sequencing

*Normal sequencing* is the absence of any other condition, and itself encompasses several addressing methods. *Micro-ECO* is the cute trick DEC uses for patching the native ROM, and is only of peripheral interest to us. A *micro-trap* is a micro-program interrupt, and a *cache stall* is what you do while waiting on a memory read. The remaining two conditions, *initialize* and *maintenance return*, do not have directly to be dealt with by the micro-programmer, but are methods by which the console computer can force the VAX micro-PC to take on certain values.

2.6.1. Normal Mode

Each microinstruction word specifies its normal successor using three fields: BEN (four bits), SUB (two bits), and JMP (thirteen bits). The latter
field carries address bits which can be used directly or in combination with
bits from other sources. BEN determines the set of conditions to be tested
when doing conditional branches, and SUB governs subroutine calls and
returns, as well as decision point jumps.

2.6.1.1. conditional branching

The destination address of a conditional branch is formed in this
manner: a set of condition bits, chosen according to field BEN, is ORed with
the contents of the JMP field; the result is used as the successor address.
The default value for BEN is zero, which specifies an all-zero set of "condi-
tions". The default value for JMP is the address of the next sequential micro-
instruction (which is unlikely to be at the next sequential address). Thus, the
default sequencing is plain, straight-line execution. The sets of condition
bits specified by BEN are shown in Figure 36. For example, to add one to the
contents of register r1, and shift the Q register left one on result zero, but to
jump to NEXT in any case, we can code:

```
ALU/A-B, AMX/LA, SPO/41, BMX/KMX, KMX/.1
; read r1, set up subtraction
; subtrahend is constant 1
--------------------------
ALU/A-B, AMX/LA, BMX/KMX, KMX/.1, SPO/51, CCK/UBCC
; do subtraction
; store result, set condition code
--------------------------
BEN/Z
; branch on Z condition (Z=1 if r1=0)
--------------------------
=0
J/NEXT
; z = 0, r1 ≠ 0
--------------------------
QK/LEFT, J/NEXT
; z = 1, r1 = 0
--------------------------
=END
```

In this example, the brackets =0 ... =END indicate that the first word within
them has an even address, and the second word the following, odd, address.
2.6.1.2. subroutines

The micro-subroutine mechanism permits subroutines and interrupts nested to a depth of sixteen. The key element is a 16x16-bit stack used to manipulate microprogram addresses. When a word in which SUB/1 is coded is executed, the address of that microword is pushed on the stack. This means that, whenever a subroutine is called, it is the address of the calling

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>FUNCT</th>
<th>UPC&lt;4&gt;</th>
<th>UPC&lt;3&gt;</th>
<th>Bits</th>
<th>UPC&lt;2&gt;</th>
<th>UPC&lt;1&gt;</th>
<th>UPC&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 NOP</td>
<td>no branch</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 Z</td>
<td>UBCC condition code</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>ALU&lt;Z&gt;</td>
<td></td>
</tr>
<tr>
<td>2 ROR</td>
<td>carry-out of ALU&lt;31&gt;</td>
<td></td>
<td></td>
<td>0</td>
<td>ALU C31</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3 C31</td>
<td>accelerator conditions</td>
<td>UB2</td>
<td></td>
<td>UB1</td>
<td>UB0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 DATA.TYPE</td>
<td>0 - normal, 1 - quad or double, 2 - field, 3 - address</td>
<td></td>
<td></td>
<td>0</td>
<td>IR&lt;1&gt;</td>
<td>IR&lt;1&gt;</td>
<td></td>
</tr>
<tr>
<td>9 IR2-1</td>
<td>IB byte zero bits</td>
<td></td>
<td></td>
<td>0</td>
<td>IR&lt;1&gt;</td>
<td>IR&lt;1&gt;</td>
<td></td>
</tr>
<tr>
<td>A REI</td>
<td>AST on REI</td>
<td>mode&lt;ASTLVL</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B IB.TEST</td>
<td>0 - TB miss, 1 - error, 2 - stall, 3 - data OK</td>
<td></td>
<td></td>
<td>0</td>
<td>IB running error/data valid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C MUL</td>
<td></td>
<td>SC ≠ 0</td>
<td>D&lt;01&gt;</td>
<td>D&lt;00&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D SIGNS</td>
<td></td>
<td>Q&lt;31&gt;</td>
<td>D ≠ 0</td>
<td>D&lt;31&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E INTERRUPT</td>
<td>test interrupt conditions</td>
<td>AC low</td>
<td>Int. inter.</td>
<td>Inter. req.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F DECIMAL</td>
<td></td>
<td>0</td>
<td>SQ&lt;7&gt;</td>
<td>D&lt;3&gt;</td>
<td>D&lt;3:0&gt; = 8</td>
<td>&amp; ≤ 39</td>
<td>or = 0D</td>
</tr>
<tr>
<td>10 UTRAP</td>
<td>micro-trap vector</td>
<td>μV&lt;3&gt;</td>
<td>μV&lt;2&gt;</td>
<td>μV&lt;1&gt;</td>
<td>μV&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 LAST.REF</td>
<td></td>
<td>μV&lt;3&gt;</td>
<td>μV&lt;2&gt;</td>
<td>μV&lt;1&gt;</td>
<td>μV&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 EALU.CC</td>
<td>EALU conditions of UBCC</td>
<td>EALU&lt;3&gt;</td>
<td>EALU&lt;Z&gt;</td>
<td>SC ≠ 0</td>
<td>SS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 SC</td>
<td>0 - zero, 1 - negative, 2 - 1-31, 3 &gt; 31</td>
<td>SC&lt;9:5&gt; = 0</td>
<td>SC&lt;9:5&gt; = 0</td>
<td>SC&lt;9:5&gt; = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 ALU1-0</td>
<td>conditions(previous cycle)</td>
<td>Log Empty</td>
<td>ALU&lt;1:0&gt; = 0</td>
<td>ALU&lt;1:0&gt; = 0</td>
<td>ALU&lt;1:0&gt; = 0</td>
<td>ALU&lt;0&gt;</td>
<td></td>
</tr>
<tr>
<td>16 STATE7-4</td>
<td>STATE register</td>
<td>STATE&lt;7&gt;</td>
<td>STATE&lt;6&gt;</td>
<td>STATE&lt;5&gt;</td>
<td>STATE&lt;4&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 STATE3-0</td>
<td>more of STATE</td>
<td>STATE&lt;3&gt;</td>
<td>STATE&lt;2&gt;</td>
<td>STATE&lt;1&gt;</td>
<td>STATE&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 D.BYTES</td>
<td>D register</td>
<td>D&lt;31:24&gt; = 0</td>
<td>D&lt;23:16&gt; = 0</td>
<td>D&lt;15:8&gt; = 0</td>
<td>D&lt;7:0&gt; = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 D3-0</td>
<td>D register</td>
<td>D&lt;3&gt;</td>
<td>D&lt;2&gt;</td>
<td>D&lt;1&gt;</td>
<td>D&lt;0&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1A PCL.CC</td>
<td>PSL conditions</td>
<td>PSL&lt;N&gt;</td>
<td>PSL&lt;Z&gt;</td>
<td>PSL&lt;V&gt;</td>
<td>PSL&lt;C&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1B ALU.CC</td>
<td>UBCC ALU conditions</td>
<td>ALU&lt;N&gt;</td>
<td>ALU&lt;Z&gt;</td>
<td>IR&lt;0&gt;</td>
<td>ALU C31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C PSL.MODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D ITB.TEST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 36 - BEN
word which is on the stack, and which must later be manipulated to form the return address. Notice how, by combining the subroutine and conditional jump mechanisms, a routine may have several entry points, conditionally chosen.

A subroutine return is denoted by SUB/2 (see Figure 37). This causes the top address on the stack to be popped and used in successor address formation; it is ORed with the logical sum of the word’s JMP field and any condition bits selected, formed as described previously. So a routine may return to any of several places relative to the calling address - depending on the JMP constant specified in the returning word, and depending on any conditions selected. I don’t believe that any overall call-return convention is used in the native microcode, as different routines tend to be called from inside diverse constrained address blocks. A simple convention, when no other address constraints are involved, is to call from an even address, and to return to the following odd address, by specifying a JMP field on 1 in the returning word.

2.6.1.3. decision point branch

When decoding operand specifiers, it is necessary to take into account information from the instruction decode logic, including operand size and addressing mode. When SUB/3 is specified, the low-order eight bits of the next micro-instruction address are taken not from any of the previously

<table>
<thead>
<tr>
<th>VALUE</th>
<th>SYMBOL</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOP</td>
<td>(default)</td>
</tr>
<tr>
<td>1</td>
<td>CALL</td>
<td>push micro-PC on stack</td>
</tr>
<tr>
<td>2</td>
<td>RET</td>
<td>OR stacktop with next address</td>
</tr>
<tr>
<td>3</td>
<td>SPEC</td>
<td>replace low 8 bits of next address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address with bits from IB</td>
</tr>
</tbody>
</table>

Figure 37 - SUB
mentioned address sources, but from the I-box. This is the SPEC jump, or
decision point branch, and is used whenever the microprogram has to use a
specifier to fetch or store an operand. Since the behavior of this operation
depends not only on the instruction stream, but on the contents of a ROM in
the instruction decode logic, it is not clear how this could be used by a
micro-program invoked by the XFC macro-instruction.

2.6.2. Micro-traps

Under certain conditions, the microprogram can be interrupted, and
control vectored to one of a set of fixed control store locations. Such a con-
trol shift should not be confused with a macro-program interrupt - some rou-
tine invoked by the micro-trap may subsequently cause a macro-processor
exception (triggered by conditions internal to the processor), but never a
macro-program interrupt (caused by external events). When micro-trap
mode is selected, a no-op cycle is performed, during which the trap address
is formed. Address bit twelve is determined by the console processor (nor-
mally zero, but permits traps to be vectored to WCS, as when running micro-
diagnostics); bit eight is set; and <3:0> comes from the exception logic, the
same as branch condition 10. The trap conditions and their associated vec-
tor addresses are shown in Figure 38; their relative priorities are given in
Figure 39. When a micro-trap is initiated, the address of the next (normal
mode) micro-instruction to be executed is pushed on the address stack, so
that the trap routine can use the subroutine return mechanism to relinquish
control. (An exception to this is the control store parity error exception,
which stacks the address of the offending word, rather than its successor. In
this case, control store may be damaged, and processing should not con-
tinue, but the address of the parity error should be reported, via the con-
2.6.3. Micro-ECO

Any machine the size of the VAX-11/780 is bound to have some initial hardware and firmware bugs. Wisely, the VAX implementors included a method of repairing the latter in a relatively cheap manner, without replacing any PROM. Microprogram changes consist of repaired code, residing in
WCS, and an FPLA, which maps PROM addresses (of erroneous code) into WCS addresses (of the repaired code). When the FPLA recognizes a bug-y address, the micro-ECO (for Engineering Change Order) logic forces a micro-no-op cycle, while the new address is formed. The micro-program continues from this address, in WCS, until it jumps back into ROM. Forty-eight such changes may be accommodated. Note that this is not a trap, of the type discussed in the last section, but only a forced change of locus, triggered by the micro-program address.