



E X T E N S I O N

## Another Approach to Instruction Set Architecture—VAX

*In principle, there is no great challenge in designing a large virtual address minicomputer system . . . . The real challenge lies in two areas: compatibility—very tangible and important; and simplicity—intangible but nonetheless important.*

William Strecker  
“VAX-11/780—A Virtual Address Extension to the PDP-11 Family,”  
*AFIPS Proc.*, National Computer Conference, 1978.

*Entities should not be multiplied unnecessarily.*

William of Occam  
Quodlibeta Septem, 1320  
(This quote is known as “Occam’s Razor.”)

III.1	Introduction	III-3
III.2	VAX Operands and Addressing Modes	III-4
III.3	Encoding VAX Instructions	III-7
III.4	VAX Operations	III-9
III.5	An Example to Put It All Together: swap	III-11
III.6	A Longer Example: sort	III-15
III.7	Fallacies and Pitfalls	III-19
III.8	Concluding Remarks	III-22
III.9	Historical Perspective and Further Reading	III-23
III.10	Exercises	III-24

---

## III.1 Introduction

The purpose of this Web extension is to give you insight into an alternative to the *Reduced Instruction Set Computer (RISC)* used in the book. To enhance your understanding of instruction set architectures, we chose the VAX as the representative *Complex Instruction Set Computers (CISC)* because it is so different from MIPS and yet still easy to understand. By seeing two such divergent styles, we are confident that you will be able to learn other instruction sets on your own.

At the time the VAX was designed, the prevailing philosophy was to create instruction sets that were close to programming languages in order to simplify compilers. For example, because programming languages had loops, instruction sets should have loop instructions. As VAX architect William Strecker said (“VAX-11/780—A Virtual Address Extension to the PDP-11 Family,” AFIPS Proc., National Computer Conference, 1978):

A major goal of the VAX-11 instruction set was to provide for effective compiler generated code. Four decisions helped to realize this goal: . . . 1) A very regular and consistent treatment of operators. . . 2) An avoidance of instructions unlikely to be generated by a compiler. . . 3) Inclusions of several forms of common operators. . . 4) Replacement of common instruction sequences with single instructions. Examples include procedure calling, multiway branching, loop control, and array subscript calculation.

Recall that DRAMs of the mid-1970s contained less than 1/1000th the capacity of today's DRAMs, so code space was also critical. Hence, another prevailing philosophy was to minimize code size, which is de-emphasized in fixed-length instruction sets like MIPS. For example, MIPS address fields always use 16 bits, even when the address is very small. In contrast, the VAX allows instructions to be a variable number of bytes, so there is little wasted space in address fields.

Books the size of the one you are reading have been written just about the VAX, so this VAX extension cannot be exhaustive. Hence, the following sections describe only a few of its addressing modes and instructions. To show the VAX instructions in action, later sections show VAX assembly code for two C procedures from Chapter 3. The general style will be to contrast these instructions with the MIPS code that you are already familiar with.

### The Big Picture

The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to the powerful addressing modes, powerful instructions, and efficient instruction encoding. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. The MIPS goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

## III.2

### VAX Operands and Addressing Modes

The VAX is a 32-bit architecture, with 32-bit wide addresses and 32-bit wide registers. Yet the VAX supports many other data sizes and types, as Figure III.1 shows. Unfortunately, VAX uses the name “word” to refer to 16-bit quantities; in this text a word means 32 bits. Figure III.1 shows the conversion between the MIPS data type names and the VAX names. Be careful when reading about VAX instructions, as they refer to the names of the VAX data types.

The VAX provides 16 32-bit registers. The VAX assembler uses the notation  $r_0, r_1, \dots, r_{15}$  to refer to these registers, and we will stick to that notation. Alas, 4 of these 16 registers are effectively claimed by the instruction set architecture. For example,  $r_{14}$  is the stack pointer ( $sp$ ) and  $r_{15}$  is the program

Bits	Data type	MIPS name	VAX name
8	Integer	Byte	Byte
16	Integer	Halfword	Word
32	Integer	Word	Long word
32	Floating point	Single precision	F_floating
64	Integer	Doubleword	Quad word
64	Floating point	Double precision	D_floating or G_floating
8n	Character string	Character	Character

FIGURE III.1 VAX data types, their lengths, and names. The first letter of the VAX type (b, w, l, f, q, d, g, c) is often used to complete an instruction name. Examples of move instructions include `movb`, `movw`, `movl`, `movf`, `movq`, `movd`, `movg`, and `movc3`. Each move instruction transfers an operand of the data type indicated by the letter following `mov`.

counter (`pc`). Hence, `r15` cannot be used as a general-purpose register, and using `r14` is very difficult because it interferes with instructions that manipulate the stack. The other dedicated registers are `r12`, used as the argument pointer (`ap`), and `r13`, used as the frame pointer (`fp`); their purpose will become clear later. (Like MIPS, the VAX assembler accepts either the register number or the register name.)

VAX addressing modes include those discussed in Chapter 3, which has all the MIPS addressing modes: *register*, *displacement*, *immediate*, and *PC-relative*. Moreover, all these modes can be used for jump addresses or for data addresses.

But that's not all the addressing modes. To reduce code size, the VAX has three lengths of addresses for displacement addressing: 8-bit, 16-bit, and 32-bit addresses called, respectively, *byte displacement*, *word displacement*, and *long displacement* addressing. Thus, an address can be not only as small as possible, but also as large as necessary; large addresses need not be split, so there is no equivalent to the MIPS `lui` instruction (see page 146).

That's still not all the VAX addressing modes. Several have a *deferred* option, meaning that the object addressed is only the *address* of the real object, requiring another memory access to get the operand. This addressing mode is called *indirect addressing* in other machines. Thus, *register deferred*, *autoincrement deferred*, and *byte/word/long displacement deferred* are other addressing modes to choose from. For example, using the notation of the VAX assembler, `r1` means the operand is register 1 and `(r1)` means the operand is the location in memory pointed to by `r1`.

There is yet another addressing mode. *Indexed addressing* automatically converts the value in an index operand to the proper byte address to add to the rest of the address. Recall the `swap` example from Chapter 3 (page 163); we needed to multiply the index of a 4-byte quantity by 4 before adding it to a base

Addressing mode name	Syntax	Example	Meaning	Length of address specifier in bytes
Literal	#value	#-1	-1	1 (6-bit signed value)
Immediate	#value	#100	100	1 + length of the immediate
Register	rn	r3	r3	1
Register deferred	(rn)	(r3)	Memory[r3]	1
Byte/word/long displacement	Displacement (rn)	100(r3)	Memory[r3 + 100]	1 + length of the displacement
Byte/word/long displacement deferred	@displacement (rn)	@100(r3)	Memory[Memory [r3 + 100]]	1 + length of the displacement
Indexed (scaled)	Base mode [rx]	(r3)[r4]	Memory[r3 + r4 × d] (where d is data size in bytes)	1 + length of base addressing mode
Autoincrement	(rn)+	(r3)+	Memory[r3]; r3 = r3 + d	1
Autodecrement	-(rn)	-(r3)	r3 = r3 - d; Memory[r3]	1
Autoincrement deferred	@(rn)+	@(r3)+	Memory[Memory[r3]]; r3 = r3 + d	1

FIGURE III.2 Definition and length of the VAX operand specifiers. The length of each addressing mode is 1 byte plus the length of any displacement or immediate field needed by the mode. Literal mode uses a special 2-bit tag and the remaining 6 bits encode the constant value. If the constant is too big, it must use the immediate addressing mode. Note the length of an immediate operand is dictated by the length of the data type indicated in the opcode, not the value of the immediate. The symbol  $d$  in the last four modes represents the length of the data in bytes;  $d$  is 4 for 32-bit add.

address. Indexed addressing, called *scaled addressing* on some computers, automatically multiplies the index of a 4-byte quantity by 4 as part of the address calculation.

To cope with such a plethora of addressing options, the VAX architecture separates the specification of the addressing mode from the specification of the operation. Hence, the opcode supplies the operation and the number of operands, and each operand has its own addressing mode specifier. Figure III.2 shows the name, assembler notation, example, meaning, and length of the address specifier.

The VAX style of addressing means that an operation doesn't know where its operands come from; a VAX `add` instruction can have three operands in registers, three operands in memory, or any combination of registers and memory operands.

**Example**

How long is the following instruction?

```
addl3 r1,737(r2),(r3)[r4]
```

The name `addl3` means a 32-bit add instruction with three operands. Assume the length of the VAX opcode is 1 byte.

**Answer**

The first operand specifier—`r1`— indicates register addressing and is 1 byte long. The second operand specifier—`737(r2)`—indicates displacement addressing and has two parts: the first part is a byte that specifies the word-displacement addressing mode and base register (`r2`); the second part is the 2-byte long displacement (`737`). The third operand specifier—`(r3)[r4]`—also has two parts: the first byte specifies register deferred addressing mode (`(r3)`), and the second byte specifies the Index register and the use of indexed addressing (`[r4]`).

Thus, the total length of the instruction is  $1 + (1) + (1+2) + (1+1) = 7$  bytes.

In this example instruction, we show the VAX destination operand on the left and the source operands on the right, just as we show MIPS code. The VAX assembler actually expects operands in the opposite order, but we felt it would be less confusing to keep the destination on the left for both machines. Obviously, left or right orientation is arbitrary; the only requirement is consistency.

Elaboration: Because the PC is one of the 16 registers that can be selected in a VAX addressing mode, 4 of the 22 VAX addressing modes are synthesized from other addressing modes. Using the PC as the chosen register in each case, *immediate* addressing is really autoincrement, *PC-relative* is displacement, *absolute* is autoincrement deferred, and *relative deferred* is displacement deferred.

**III.3****Encoding VAX Instructions**

Given the independence of the operations and addressing modes, the encoding of instructions is quite different from MIPS.

VAX instructions begin with a single byte opcode containing the operation and the number of operands. The operands follow the opcode. Each operand begins with a single byte, called the *address specifier*, that describes the addressing mode for that operand. For a simple addressing mode, such as register

Byte address	Contents at each byte	Machine code
201	opcode containing <code>addl3</code>	<code>c1<sub>hex</sub></code>
202	index mode specifier for <code>[r4]</code>	<code>44<sub>hex</sub></code>
203	register indirect mode specifier for <code>(r3)</code>	<code>63<sub>hex</sub></code>
204	word displacement mode specifier using <code>r2</code> as base	<code>c2<sub>hex</sub></code>
205	the 16-bit constant <code>737</code>	<code>e1<sub>hex</sub></code>
206		<code>02<sub>hex</sub></code>
207	register mode specifier for <code>r1</code>	<code>51<sub>hex</sub></code>

FIGURE III.3 The encoding of the VAX instruction `addl3 r1,737(r2),(r3)[r4]`, assuming it starts at address 201. To satisfy your curiosity, the right column shows the actual VAX encoding in hexadecimal notation (page 158 describes hexadecimal notation). Note that the 16-bit constant `737ten` takes two bytes.

addressing, this byte specifies the register number as well as the mode (see the rightmost column in Figure III.2). In other cases, this initial byte can be followed by many more bytes to specify the rest of the address information.

As a specific example, let's show the encoding of the `add` instruction from the example on page III-7:

```
addl3 r1,737(r2),(r3)[r4]
```

Assume that this instruction starts at location 201.

Figure III.3 shows the encoding. Note that the operands are stored in memory in opposite order to the assembly code above. The execution of VAX instructions begins with fetching the source operands, so it makes sense for them to come first. Order is not important in fixed-length instructions like MIPS, since the source and destination operands are easily found within a 32-bit word.

The first byte, at location 201, is the opcode. The next byte, at location 202, is a specifier for the index mode using register `r4`. Like many of the other specifiers, the left 4 bits of the specifier give the mode and the right 4 bits give the register used in that mode. Since `addl3` is a 4-byte operation, `r4` will be multiplied by 4 and added to whatever address is specified next. In this case it is register deferred addressing using register `r3`. Thus bytes 202 and 203 combined define the third operand in the assembly code.

The following byte, at address 204, is a specifier for word displacement addressing using register `r2` as the base register. This specifier tells the VAX that the following two bytes, locations 205 and 206, contain a 16-bit address to be added to `r2`.

The final byte of the instruction gives the destination operand, and this specifier selects register addressing using register `r1`.

Such variability in addressing means that a single VAX operation can have many different lengths; for example, an integer add varies from 3 bytes to 19 bytes. VAX implementations must decode the first operand before they can find the second, and so implementors are strongly tempted to take one clock cycle to decode each operand; thus this sophisticated instruction set architecture can result higher clock cycles per instruction, even when using simple addresses.

## III.4

### VAX Operations

In keeping with its philosophy, the VAX has a large number of operations as well as a large number of addressing modes. We review a few here to give the flavor of the machine.

Given the power of the addressing modes, the VAX *move* instruction performs several operations found in other machines. It transfers data between any two addressable locations and subsumes load, store, register–register moves, and memory–memory moves as special cases. The first letter of the VAX data type (b, w, l, f, q, d, g, c in Figure III.1) is appended to the acronym `mov` to determine the size of the data. One special move, called *move address*, moves the 32-bit *address* of the operand rather than the data. It uses the acronym `mov a`.

The arithmetic operations of MIPS are also found in the VAX, with two major differences. First, the type of the data is attached to the name. Thus `addb`, `addw`, and `addl` operate on 8-bit, 16-bit, and 32-bit data in memory or registers, respectively; MIPS has a single `add` instruction that operates only on the full 32-bit register. The second difference is that to reduce code size, the `add` instruction specifies the number of unique operands; MIPS always specifies three even if one operand is redundant. For example, the MIPS instruction

```
add    $1, $1, $2
```

takes 32 bits like all MIPS instructions, but the VAX instruction

```
addl2 r1, r2
```

uses `r1` for both the destination and a source, taking just 24 bits: 8 bits for the opcode and 8 bits each for the two register specifiers.

#### Number of Operations

Now we can show how VAX instruction names are formed:

$$(\text{operation})(\text{datatype}) \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

The operation `add` works with data types `byte`, `word`, `long`, `float`, and `double` and comes in versions for either 2 or 3 unique operands, so the following instructions are all found in the VAX:

```

addb2    addw2    addl2    addf2    addd2
addb3    addw3    addl3    addf3    addd3

```

Accounting for all addressing modes (but ignoring register numbers and immediate values) and limiting to just `byte`, `word`, and `long`, there are more than 30,000 versions of integer `add` in the VAX; MIPS has just 4!

Another reason for the large number of VAX instructions is the instructions that either replace sequences of instructions or take fewer bytes to represent a single instruction. Here are four such examples (\* means the data type):

VAX operation	Example	Meaning
<code>clr*</code>	<code>clr1 r3</code>	$r3 = 0$
<code>inc*</code>	<code>incl r3</code>	$r3 = r3 + 1$
<code>dec*</code>	<code>decl r3</code>	$r3 = r3 - 1$
<code>push*</code>	<code>push1 r3</code>	$sp = sp - 4$ ; $\text{Memory}[sp] = r3$ ;

The `push` instruction in the last row is exactly the same as using the `move` instruction with autodecrement addressing on the stack pointer:

```
mov1 -(sp), r3
```

Brevity is the advantage of `push1`: it is one byte shorter since `sp` is implied.

## Branches, Jumps, and Procedure Calls

The VAX branch instructions are related to the arithmetic instructions because the branch instructions rely on *condition codes*. Condition codes are set as a side effect of an operation, and they indicate whether the result is positive, negative, zero, or if an overflow occurred (see page 181 in Chapter 4). Most instructions set the VAX condition codes according to their result; instructions without results, such as branches, do not. The VAX condition codes are N (Negative), Z (Zero), V (oVerflow), and C (Carry). There is also a *compare* instruction `cmp*` just to set the condition codes for a subsequent branch.

The VAX branch instructions include all conditions. Popular branch instructions include `beql(=)`, `bneq(≠)`, `blss(<)`, `bleq(≤)`, `bgtr(>)`, and `bgeq(≥)`, which do just what you would expect. There are also unconditional branches whose name is determined by the size of the PC-relative offset. Thus `brb` (*branch byte*) has an 8-bit displacement and `brw` (*branch word*) has a 16-bit displacement.

The final major category we cover here is the procedure *call and return* instructions. Unlike the MIPS architecture, these elaborate instructions can take dozens of clock cycles to execute. The next two sections show how they work, but we need to explain the purpose of the pointers associated with the stack manipulated by `calls` and `ret`. The *stack pointer*, `sp`, is just like the stack pointer in MIPS; it points to the top of the stack. The *argument pointer*, `ap`, points to the base of the list of arguments or parameters in memory that are passed to the procedure. The *frame pointer*, `fp`, points to the base of the local variables of the procedure that are kept in memory (the *stack frame*). The VAX call and return instructions manipulate these pointers to maintain the stack in proper condition across procedure calls and to provide convenient base registers to use when accessing memory operands. As we shall see, call and return also save and restore the general purpose registers as well as the program counter.

Figure III.4 gives a further sampling of the VAX instruction set.

## III.5

### An Example to Put It All Together: swap

To see programming in VAX assembly language, we translate the C procedures `swap` and `sort` from Chapter 3; the C code for `swap` is reproduced in Figure III.5. The next section covers `sort`.

Just as we did in section 3.10 of Chapter 3, we describe the `swap` procedure in these three general steps of assembly language programming:

1. Allocate registers to program variables
2. Produce code for the body of the procedure
3. Preserve registers across the procedure invocation

The VAX code for these procedures is based on code produced by the VMS C compiler using optimization.

#### Register Allocation for *swap*

In contrast to MIPS, VAX parameters are normally allocated to memory, so this step of assembly language programming is more properly called “variable allocation.” The standard VAX convention on parameter passing is to use the stack. The two parameters, `v[]` and `k`, can be accessed using register `ap`, the argument pointer: the address  $4(ap)$  corresponds to `v[]` and  $8(ap)$  corresponds to `k`. Remember that with byte addressing the address of sequential 4-byte words differs by 4. The only other variable is `temp`, which we associate with register `r3`.

Instruction type	Example	Instruction meaning
Data transfers		Move data between byte, halfword, word, or doubleword operands; * is data type
	mov*	Move between two operands
	movzb*	Move a byte to a halfword or word, extending it with zeroes
	mova*	Move the 32-bit address of an operand; data type is last
	push*	Push operand onto stack
Arithmetic, logical		Operations on integer or logical bytes, halfwords (16 bits), words (32 bits); * is data type
	add*_	Add with 2 or 3 operands
	cmp*	Compare and set condition codes
	tst*	Compare to zero and set condition codes
	ash*	Arithmetic shift
	clr*	Clear
	cvtb*	Sign-extend byte to size of data type
Control		Conditional and unconditional branches
	beql, bneq	Branch equal, branch not equal
	bleq, bgeq	Branch less than or equal, branch greater than or equal
	brb, brw	Unconditional branch with an 8-bit or 16-bit address
	jmp	Jump using any addressing mode to specify target
	aobleq	Add one to operand; branch if result $\leq$ second operand
	case_	Jump based on case selector
Procedure		Call/return from procedure
	calls	Call procedure with arguments on stack (see section 6)
	callg	Call procedure with FORTRAN-style parameter list
	jsb	Jump to subroutine, saving return address (like MIPS jal)
	ret	Return from procedure call
Floating point		Floating-point operations on D, F, G, and H formats
	addd_	Add double-precision D-format floating numbers
	subd_	Subtract double-precision D-format floating numbers
	mulf_	Multiply single-precision F-format floating point
	polyf	Evaluate a polynomial using table of coefficients in F format
Other		Special operations
	crc	Calculate cyclic redundancy check
	insque	Insert a queue entry into a queue

FIGURE III.4 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in `addd_`, means there are 2-operand (`addd2`) and 3-operand (`addd3`) forms of this instruction.

```

swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

```

FIGURE III.5 A C procedure that swaps two locations in memory. This code is a copy of Figure 3.23 on page 164. This procedure will be used in the sorting example in the next section. Web Extension II shows the C and Pascal versions of this procedure side-by-side (page II-5).

### Code for the Body of the Procedure swap

The remaining lines of C code in `swap` are

```

temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;

```

Since this program uses `v[]` and `k` several times, to make the programs run faster the VAX compiler first moves both parameters into registers:

```

movl  r2, 4(ap); r2 = v[]
movl  r1, 8(ap); r1 = k

```

Note that we follow the VAX convention of using a semicolon to start a comment; the MIPS comment symbol `#` represents a constant operand in VAX assembly language.

The VAX has indexed addressing, so we can use index `k` without converting it to a byte address. The VAX code is then straightforward:

```

movl  r3, (r2)[r1]      ; r3 (temp) = v[k]
addl3 r0, #1,8(ap)     ; r0 = k + 1
movl  (r2)[r1],(r2)[r0] ; v[k] = v[r0] (v[k + 1])
movl  (r2)[r0],r3      ; v[k + 1] = r3 (temp)

```

Unlike the MIPS code, which is basically two loads and two stores, the key VAX code is one memory-to-register move, one memory-to-memory move, and one register-to-memory move. Note that the `addl3` instruction shows the flexibility of the VAX addressing modes: It adds the constant 1 to a memory operand and places the result in a register.

Now we have allocated storage and written the code to perform the operations of the procedure. The only missing item is the code that preserves registers across the routine that calls `swap`.

## Preserving Registers Across Procedure Invocation of swap

The VAX has a pair of instructions that preserve registers `calls` and `ret`. This example shows how they work.

The VAX C compiler uses a form of callee convention. Examining the code above, we see that the values in registers `r0`, `r1`, `r2`, and `r3` must be saved so that they can later be restored. The `calls` instruction expects a 16-bit mask at the beginning of the procedure to determine which registers are saved: if bit  $i$  is set in the mask, then register  $i$  is saved on the stack by the `calls` instruction. In addition, `calls` saves this mask on the stack to allow the return instruction (`ret`) to restore the proper registers. Thus the `calls` executed by the caller does the saving, but the callee sets the call mask to indicate what should be saved.

One of the operands for `calls` gives the number of parameters being passed, so that `calls` can adjust the pointers associated with the stack: the argument pointer (`ap`), frame pointer (`fp`), and stack pointer (`sp`). Of course, `calls` also saves the program counter so that the procedure can return!

Thus, to preserve these four registers for `swap`, we just add the mask at the beginning of the procedure, letting the `calls` instruction in the caller do all the work:

```
.word ^m<r0,r1,r2,r3>; set bits in mask for 0,1,2,3
```

This directive tells the assembler to place a 16-bit constant with the proper bits set to save registers `r0` through `r3`.

The return instruction undoes the work of `calls`. When finished, `ret` sets the stack pointer from the current frame pointer to pop everything `calls` placed on the stack. Along the way, it restores the register values saved by `calls`, including those marked by the mask and old values of the `fp`, `ap`, and `pc`.

To complete the procedure `swap`, we just add one instruction:

```
ret ; restore registers and return
```

## The Full Procedure swap

We are now ready for the whole routine. Figure III.6 identifies each block of code with its purpose in the procedure, with the MIPS code on the left and the VAX code on the right. This example shows the advantage of the scaled indexed addressing and the sophisticated call and return instructions of the VAX in reducing the number of lines of code. The 17 lines of MIPS assembly code became 8 lines of VAX assembly code. It also shows that passing parameters in memory results in extra memory accesses.

Keep in mind that the number of instructions executed is not the same as performance; the fallacies on pages 185 and III-21 make this point.

MIPS versus VAX	
Saving register	
<pre>swap: addi \$29,\$29, -12       sw   \$2, 0(\$29)       sw   \$15, 4(\$29)       sw   \$16, 8(\$29)</pre>	<pre>swap: .word ^m&lt;r0,r1,r2,r3&gt;</pre>
Procedure body	
<pre>      muli \$2, \$5,4       add  \$2, \$4,\$2       lw   \$15, 0(\$2)       lw   \$16, 4(\$2)       sw   \$16, 0(\$2)       sw   \$15, 4(\$2)</pre>	<pre>      movl r2, 4(a)       movl r1, 8(a)       movl r3, (r2)[r1]       addl3 r0, #1,8(ap)       movl (r2)[r1],(r2)[r0]       movl (r2)[r0],r3</pre>
Restoring registers	
<pre>      lw   \$2, 0(\$29)       lw   \$15, 4(\$29)       lw   \$16, 8(\$29)       addi \$29,\$29, 12</pre>	
Procedure return	
<pre>      jr   \$31</pre>	<pre>      ret</pre>

FIGURE III.6 MIPS versus VAX assembly code of the procedure swap in Figure III.5 on page III-13.

Elaboration: VAX software follows a convention of treating registers `r0` and `r1` as temporaries which are not saved across a procedure call, so the VMS C compiler does include registers `r0` and `r1` in the register saving mask. Also, the C compiler should have used `r1` instead of `8(ap)` in the `addl3` instruction; such examples inspire computer architects to try to write compilers!

## III.6 A Longer Example: sort

As in Chapter 3, we show the longer example of the `sort` procedure. Figure III.7 shows the C version of the program. Once again we present this procedure in several steps, concluding with a side-by-side comparison to MIPS code.

### Register Allocation for sort

The two parameters of the procedure `sort`, `v` and `n`, are found in the stack in locations `4(ap)` and `8(ap)`, respectively. The two local variables are assigned

```

sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i = i + 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1) { swap(v,j);
        }
    }
}

```

FIGURE III.7 A C procedure that performs a bubble sort on the array *v*. This code is a copy of Figure 3.25 on page 166. (See Web Extension II for a Pascal version of *sort*.)

to registers: *i* to *r6* and *j* to *r4*. Because the two parameters are referenced frequently in the code, the VMS C compiler copies the *address* of these parameters into registers upon entering the procedure:

```

movl r7,8(ap) ; move address of n into r7
movl r5,4(ap) ; move address of v into r5

```

It would seem that moving the *value* of the operand to a register would be more useful than its address, but once again we bow to the decision of the VMS C compiler. Apparently the compiler cannot be sure that *v* and *n* don't overlap in memory.

### Code for the Body of the sort Procedure

The procedure body consists of two nested *for* loops and a call to *swap* which includes parameters. Let's unwrap the code from the outside to the middle.

#### The Outer Loop

The first translation step is the first *for* loop:

```

for (i = 0; i < n; i = i+1) {

```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize *i* to 0, the first part of the *for* statement:

```

    clrl r6 ; i = 0

```

It also takes just one instruction to increment *i*, the last part of the *for*:

```

    incl r6 ; i = i + 1

```

The loop should be exited if *i* < *n* is *false*, or said another way, exit the loop if *i* ≥ *n*. This test takes two instructions:

```

for1tst: cml r6,(r7) ; compare r6 and memory[r7] (i:n)
          bgeq exit1 ; go to exit1 if r6 ≥ mem[r7] (i ≥ n)

```

Note that `cmpl` sets the condition codes for use by the conditional branch instruction `bgeq`.

The bottom of the loop just jumps back to the loop test:

```
        brb  for1tst ; branch to test of outer loop
exit1:
```

The skeleton code of the first *for* loop is then:

```
        clr1 r6      ; i = 0
for1tst: cmpl r6,(r7) ; compare r6 and memory[r7] (i:n)
        bgeq exit1  ; go to exit1 if r6 ≥ mem[r7] (i ≥ n)
        ...
        (body of first for loop)
        ...
        incl r6     ; i = i + 1
        brb  for1tst ; branch to test of outer loop
exit1:
```

Exercise 3.9 on page 199 explores writing faster code for the similar loops.

The Inner Loop

The second *for* loop is

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1) {
```

The initialization portion of this loop is again one instruction:

```
        subl3 r4,r6,#1 ; j = i - 1
```

and the decrement of *j* is also one instruction:

```
        decl  r4      ; j = j - 1
```

The loop test has two parts. We exit the loop if either condition fails, so first test must exit the loop if it fails ( $j < 0$ ):

```
for2tst:blss exit2 ; go to exit2 if r4 < 0 (j < 0)
```

Notice that there is no explicit comparison. The lack of comparison is a benefit of condition codes, with the conditions being set as a side effect of the prior instruction. This branch skips over the second condition test.

The second test exits if  $v[j] > v[j + 1]$  is *false*, or exit if  $v[j] \leq v[j + 1]$ .

First we load *v* and put *j* + 1 into registers:

```
        movl  r3,(r5) ; r3 = Memory[r5] (r3 = v)
        addl3 r2,r4,#1 ; r2 = r4 + 1 (r2 = j + 1)
```

Register indirect addressing is used to get the operand pointed to by *r5*.

Once again the index addressing mode means we can use indices without converting to the byte address, so the two instructions for  $v[j] \leq v[j + 1]$  are

```
        cmpl  (r3)[r4],(r3)[r2] ; v[r4] : v[r2] (v[j]:v[j + 1])
        bleq  exit2             ; go to exit2 if v[j] ≤ v[j + 1]
```

The bottom of the loop jumps back to the full loop test:

```
brb for2tst # jump to test of inner loop
```

Combining the pieces, the second *for* loop looks like this:

```

subl3 r4,r6,#1 ; j = i - 1
for2tst: blss exit2 ; go to exit2 if r4 < 0 (j < 0)
movl r3,(r5) ; r3 = Memory[r5] (r3 = v)
addl3 r2,r4,#1 ; r2 = r4 + 1 (r2 = j + 1)
cmpl (r3)[r4],(r3)[r2]; v[r4] : v[r2]
bleq exit2 ; go to exit2 if v[j]≤v[j+1]
...
; (body of second for loop)
...
decl r4 ; j = j - 1
brb for2tst ; jump to test of inner loop
exit2:
```

Notice that the instruction `blss` (at the top of the loop) is testing the condition codes based on the new value of `r4` (`j`), set either by the `subl3` before entering the loop or by the `decl` at the bottom of the loop.

The Procedure Call

The next step is the body of the second *for* loop:

```
swap(v,j);
```

Calling `swap` is easy enough:

```
calls #2,swap
```

The constant `2` indicates the number of parameters pushed on the stack.

Passing Parameters

The C compiler passes variables on the stack, so we pass the parameters to `swap` with these two instructions:

```

pushl (r5) ; first swap parameter is v
pushl r4 ; second swap parameter is j
```

Register indirect addressing is used to get the operand of the first instruction.

Preserving Registers Across Procedure Invocation of `sort`

The only remaining code is the saving and restoring of registers using the callee save convention. This procedure uses registers `r2` through `r7`, so we add a mask with those bits set:

```
.word ^m<r2,r3,r4,r5,r6,r7>; set mask for registers 2-7
```

Since `ret` will undo all the operations, we just tack it on the end of the procedure.

### The Full Procedure `sort`

Now we put all the pieces together in Figure III.8. To make the code easier to follow, once again we identify each block of code with its purpose in the procedure and list the MIPS and VAX code side-by-side. In this example, 11 lines of the `sort` procedure in C become the 44 lines in the MIPS assembly language and 20 lines in VAX assembly language. The biggest VAX advantages are in register saving and restoring and indexed addressing.

Elaboration: The optimizing VMS C compiler did several tricks to improve this code, including replacing the call of the `swap` procedure with the body of the code inside the `sort` procedure, thereby avoiding the overhead of procedure call and return. Actually, the MIPS C compiler uses a much more efficient register save/restore convention than the one shown in Figure III.8, so the number of lines of code for each architecture is much closer than the figure suggests. Both compilers also use more efficient loops. We show them in this form to make the code easier to follow.

## III.7

### Fallacies and Pitfalls

*The ability to simplify means to eliminate the unnecessary so that the necessary may speak.*

Hans Hoffman, *Search for the Real*, 1967

*Fallacy: It is possible to design a flawless architecture.*

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at one time later look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code-size efficiency and underestimated how important ease of decoding and pipelining would be ten years later. And almost all architectures eventually succumb to the lack of sufficient address space. Avoiding these problems in the long run, however, would probably mean compromising the efficiency of the architecture in the short run.

MIPS versus VAX		
Saving registers		
	<pre> sort:  addi \$29,\$29, -36         sw  \$15, 0(\$29)         sw  \$16, 4(\$29)         sw  \$17, 8(\$29)         sw  \$18,12(\$29)         sw  \$19,16(\$29)         sw  \$20,20(\$29)         sw  \$24,24(\$29)         sw  \$25,28(\$29)         sw  \$31,32(\$29)                     </pre>	<pre> sort:  .word ^m&lt;r2,r3,r4,r5,r6,r7&gt;                     </pre>
Procedure body		
Move parameters	<pre> move \$18, \$4 move \$20, \$5                     </pre>	<pre> movl  r7,8(ap) movl  r5,4(ap)                     </pre>
Outer loop	<pre> for1tst: add \$19, \$0, \$0         slt \$8, \$19, \$20         beq \$8, \$0, exit1                     </pre>	<pre> for1tst: clr  r6         cpl  r6,(r7)         bgeq exit1                     </pre>
Inner loop	<pre> for2tst: addi \$17, \$19, -1         slti \$8, \$17, 0         bne \$8, \$0, exit2         muli \$15, \$17, 4         add \$16, \$18, \$15         lw  \$24, 0(\$16)         lw  \$25, 4(\$16)         slt \$8, \$25, \$24         beq \$8, \$0, exit2                     </pre>	<pre> for2tst: subl3 r4,r6,#1         blss exit2         movl r3,(r5)          addl3 r2,r4,#1         cpl  (r3)[r4],(r3)[r2]         bleq exit2                     </pre>
Pass parameters and call	<pre> move \$4, \$18 move \$5, \$17 jal  swap                     </pre>	<pre> pushl (r5) pushl r4 calls #2,swap                     </pre>
Inner loop	<pre> addi \$17, \$17, -1 j    for2tst                     </pre>	<pre> decl  r4 brb  for2tst                     </pre>
Outer loop	<pre> exit2:  addi \$19, \$19, 1         j    for1tst                     </pre>	<pre> exit2:  incl  r6         brb  for1tst                     </pre>
Restoring registers		
	<pre> exit1:  lw  \$15, 0(\$29)         lw  \$16, 4(\$29)         lw  \$17, 8(\$29)         lw  \$18,12(\$29)         lw  \$19,16(\$29)         lw  \$20,20(\$29)         lw  \$24,24(\$29)         lw  \$25,28(\$29)         lw  \$31,32(\$29)         addi \$29,\$29, 36                     </pre>	
Procedure return		
	<pre> jr  \$31                     </pre>	<pre> exit1:  ret                     </pre>

FIGURE III.8 MIPS versus VAX assembly version of procedure sort in Figure III.7 on page III-16.

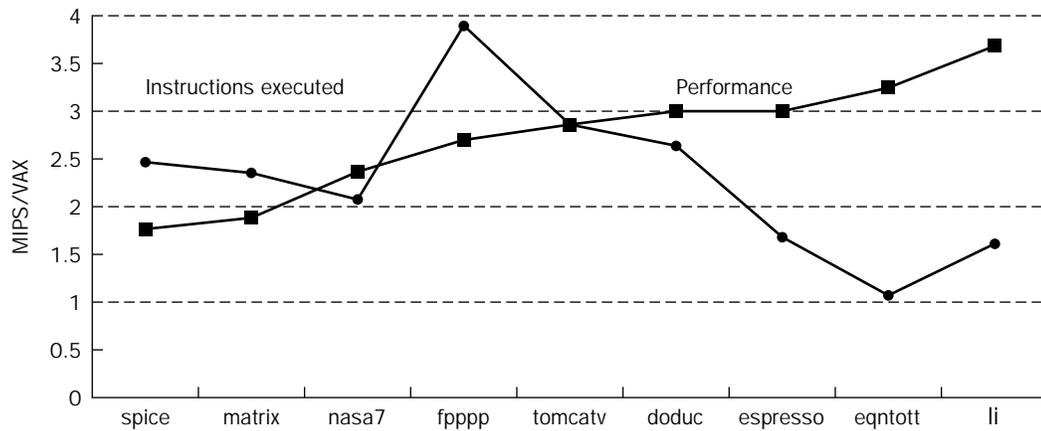


FIGURE III.9 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from “Performance from Architecture: Comparing a RISC and CISC with Similar Hardware Organization,” by D. Bhandarkar and D. Clark in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems IV*, 1991.)

*Fallacy: An architecture with flaws cannot be successful.*

The IBM 360 is often criticized in the literature—the branches are not PC-relative, and the address is too small in displacement addressing. Yet, the machine has been an enormous success because it correctly handled several new problems. First, the architecture has a large amount of address space. Second, it is byte addressed and handles bytes well. Third, it is a general-purpose register machine. Finally, it is simple enough to be efficiently implemented across a wide performance and cost range.

The Intel 8086 provides an even more dramatic example. The 8086 architecture is the only widespread architecture in existence today that is not truly a general-purpose register machine. Furthermore, the segmented address space of the 8086 causes major problems both for programmers and compiler writers. Finally, it is hard to implement. It has generally provided only half the performance of the RISC architectures for the last eight years, despite significant investment by Intel. Nevertheless, the 8086 architecture—because of its selection as the microprocessor in the IBM PC—has been enormously successful.

*Fallacy: The architecture that executes fewer instructions is faster.*

Designers of VAX machines performed a quantitative comparison of VAX and MIPS for implementations with comparable organizations, the VAX 8700 and the MIPS M2000. Figure III.9 shows the ratio of the number of instructions

executed and the ratio of performance measured in clock cycles. MIPS executes about twice as many instructions as the VAX while the MIPS M2000 has almost three times the performance of the VAX 8700.

## III.8

### Concluding Remarks

*The Virtual Address eXtension of the PDP-11 architecture . . . provides a virtual address of about 4.3 gigabytes which, even given the rapid improvement of memory technology, should be adequate far into the future.*

William Strecker, "VAX-11/780—A Virtual address Extension to the PDP-11 Family," *AFIPS Proc., National Computer Conference*, 1978

We have seen that instruction sets can vary quite dramatically, both in how they access operands and in the operations that can be performed by a single instruction. Figure III.10 compares instruction usage for both architectures for two programs; even very different architectures behave similarly in their use of instruction classes.

A product of its time, the VAX emphasis on code density and complex operations and addressing modes conflicts with the current emphasis on easy decoding, simple operations and addressing modes, and pipelined performance.

With more than 600,000 sold, the VAX architecture has had a very successful run. DEC has made the transition from VAX to Alpha, a 64-bit address architecture very similar to MIPS.

#### The Big Picture

Orthogonality is key to the VAX architecture; the opcode is independent of the addressing modes that are independent of the data types and even the number of unique operands. Thus a few hundred operations expand to hundreds of thousands of instructions when accounting for the data types, operand counts, and addressing modes.

Program	Machine	Branch	Arithmetic logical	Data transfer	Floating point	Totals
gcc	VAX	30%	40%	19%		89%
	MIPS	24%	35%	27%		86%
spice	VAX	18%	23%	15%	23%	79%
	MIPS	4%	29%	35%	15%	83%

FIGURE III.10 The frequency of instruction distribution for two programs on VAX and MIPS.

## III.9

### Historical Perspective and Further Reading

*VAX: the most successful minicomputer design in industry history . . . the VAX was probably the hacker's favorite machine . . . Especially noted for its large, assembler-programmer-friendly instruction set—an asset that became a liability after the RISC revolution.*

Eric Raymond, *The New Hacker's Dictionary*, 1991

In the mid-1970s, DEC realized that the PDP-11 was running out of address space. The 16-bit space had been extended in several creative ways, but the small address space was a problem that could only be postponed, not overcome.

In 1977, DEC introduced the VAX. Strecker described the architecture and called the VAX “a Virtual Address eXtension of the PDP-11.” One of DEC’s primary goals was to keep the installed base of PDP-11 customers. Thus, the customers were to think of the VAX as a 32-bit successor to the PDP-11. A 32-bit PDP-11 was possible—there were three designs—but Strecker reports that they were “overly compromised in terms of efficiency, functionality, programming ease.” The chosen solution was to design a new architecture and include a PDP-11 compatibility mode that would run PDP-11 programs without change. This mode also allowed PDP-11 compilers to run and to continue to be used. The VAX-11/780 resembled the PDP-11 in many ways. These are among the most important:

1. Data types and formats are mostly equivalent to those on the PDP-11. The F and D floating formats came from the PDP-11. G and H formats were added later. The use of the term “word” to describe a 16-bit quantity was carried from the PDP-11 to the VAX.
2. The assembly language was made similar to the PDP-11s.
3. The same buses were supported (Unibus and Massbus).

4. The operating system, VMS, was “an evolution” of the RSX-11M/IAS OS (as opposed to the DECsystem 10/20 OS, which was a more advanced system), and the file system was basically the same.

The VAX-11/780 was the first machine announced in the VAX series. It is one of the most successful and heavily studied machines ever built. It relied heavily on microprogramming (Chapter 5), taking advantage of the increasing capacity of fast semiconductor memory to implement the complex instructions and addressing modes. The VAX is so tied to microcode that we predict it will be impossible to build the full VAX instruction set without microcode.

To offer a single-chip VAX in 1984, DEC reduced the instructions interpreted by microcode by trapping some instructions and performing them in software. DEC engineers found that 20% of VAX instructions are responsible for 60% of the microcode, yet are only executed 0.2% of the time. The final result was a chip offering 90% of the performance with a reduction in silicon area by more than a factor of five.

The cornerstone of DEC’s strategy was a single architecture, VAX, running a single operating system, VMS. This strategy worked well for over ten years. Today, DEC has transitioned to the Alpha RISC architecture. Like the transition from the PDP-11 to the VAX, Alpha offers the same operating system, file system, and data types and formats of the VAX. Instead of providing a VAX compatibility mode, the Alpha approach is to “compile” the VAX machine code into the Alpha machine code.

### To Probe Further

Levy, H., and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

*This book concentrates on the VAX, but includes descriptions of other machines.*

## III.10

### Exercises

III.1 [3] <§§3.2, 3.10, III.4> The following VAX instruction decrements the location pointed to be register r5:

```
decl (r5)
```

What is the single MIPS instruction, or if it cannot be represented in a single instruction, the shortest sequence of MIPS instructions, that performs the same operation? What are the lengths of the instructions on each machine?

III.2 [5] <§§3.2, 3.10, III.4> This exercise is the same as Exercise III.1, except this VAX instruction clears a location using autoincrement deferred addressing:

```
clr1 @(r5)+
```

III.3 [5] <§§3.2, 3.5, III.5> This exercise is the same as Exercise III.1, except this VAX instruction adds 1 to register r5, placing the sum back in register r5, compares the sum to register r6, and then branches to L1 if r5 < r6:

```
aoblss r6, r5,L1 # r5 = r5 + 1; if (r5 < r6) goto L1.
```

III.4 [5] <§III.2> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
a = b + 100;
```

Assume a corresponds to register r3 and b corresponds to register r4.

III.5 [10] <§III.2> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
x[i + 1] = x[i] + c;
```

Assume c corresponds to register r3, i to register r4, and x is an array of 32-bit words beginning at memory location 4,000,000<sub>ten</sub>.



