15

Turning Cousins into Sisters: An Example of Software Smoothing of Hardware Differences

RONALD F. BRENDER

INTRODUCTION

In 1970, the PDP-11 was Digital Equipment Corporation's newly announced minicomputer and its first offering in the 16-bit world. Among the many software components needed to complement the hardware, a FORTRAN system was high on the list. A FORTRAN project was begun in 1970 and the first release of the resulting product took place in mid-1971. In the succeeding years, the number of PDP-11 CPUs and related options increased dramatically to provide a wide range of price/performance alternatives. What makes the original FORTRAN interesting, even today, is the extent to which the basic implementation approach was able to be extended gracefully to span the entire family with modest incremental effort.

This paper describes the design concepts, threaded code and a FORTRAN virtual machine, used to implement the original PDP-11 FORTRAN product. As the PDP-11 family of processors expanded with new models and options, these original design concepts proved both stable enough and flexible enough to be employed successfully across the entire family.

When this FORTRAN was finally superseded in early 1975, it had two successors. One, called FORTRAN IV, continued the threaded code and virtual machine concepts of the earlier product with similar execution performance across the PDP-11 family, but offered much faster compilation rates in smaller memory. The other successor, called FORTRAN IV-PLUS, produced direct PDP-11 code and obtained significantly improved execution performance for the PDP-11/45, PDP-11/70, and PDP-11/60 with FP11 floating-point hardware relative to both of the other FORTRANs.

In the Beginning

The PDP-11/20 was a significant advance over other minicomputers of its time, but was a bare machine architecture by today's standards. There was no floating-point hardware of any kind (even as an option) and integer multiply and divide operations were available only by means of an I/O bus option, the Extended Arithmetic Element (EAE). (The EAE also provided multiple-bit arithmetic shift operations;

the PDP-11/20 instructions provided only single-bit shifts.)

The first disk-based operating system, DOS, was designed for a minimum standard system that included 8 Kwords (16 Kbytes) of memory. After allowing typically 2 Kwords for the resident parts of the monitor, only 5 K to 6 K remained for other use. Consequently, size constraints played a major role in the FORTRAN system design and implementation.

There were not many competitors at the time, but at least one, the IBM 1130, offered a disk-based operating system and FORTRAN system. To meet this competition, an important goal was to deliver the PDP-11 FORTRAN system to the market as quickly as possible, even at the cost of performance, if necessary.

Neither Compiler nor Interpreter, but Threaded Code

The fundamental design strategy to be determined was the structure of the executing code, the "run-time environment" [DEC, 1974b; DEC, 1974c].

We were leery of a compiler that generated direct machine code primarily because of the size of compiled code. Much of the compiled code would necessarily consist of calls to floating-point and other support routines, and on the PDP-11, each subroutine call required two words of memory, not counting argument transmission.

An interpreter would easily solve the space problem, but this had its own disadvantages. The basic interpreter loop overhead was a concern, but not crucial at that stage in our deliberations. However, a disadvantage of interpreters is that they must be "always present" even though not all of the capabilities are being used. For example, routines for complex arithmetic are part of the interpreter even though the particular program in use does not perform complex arithmetic. Further, we wanted to maintain the traditional FORTRAN features of independent compilation and linking of routines, and easy writing of routines in assembler for inclusion in the program.

The solution was threaded code [Bell, J., 1973]. Threaded code is a kind of combination of an interpreter and compiled code with most of the best features of each. On the PDP-11 it works in the following way.

The "compiled code" consists simply of a sequence of service routine addresses. A single register (we used R4) is chosen to contain a pointer to the next address in the sequence to be invoked. Each service routine completes by transferring control to the next routine in the sequence and simultaneously advancing the pointer.

To illustrate, consider a service routine whose purpose is to perform floating-point addition of two real values found in a stack (we used R6, the hardware stack pointer, for the value stack) and leave the result on the top of the stack in place of the parameters. The service routine would look like the following.*

```
$ADR: <<code for floating point add>>
      JMP @(R4)+
```

The JMP instruction with deferred auto-increment addressing mode provides just the

*The brackets << and >> are used in examples in place of code to indicate the purpose of code that is too bulky and/or not relevant for the example.

In the PDP-11 MACRO assembler language [DEC, 1976], identifiers may consist of up to six characters from among the letters, numerals, "." and "$". Identifiers created by the FORTRAN compiler include either a period or dollar sign to assure that they are distinct from FORTRAN language identifiers.

In the PDP-11 MACRO assembler language, a colon follows a label and separates the label from assembler instructions.

combination needed to sequence through the table of addresses. It is a single one-word instruction.

The instruction corresponds to the basic loop of an interpreter. Consequently, there is no centralized interpreter: the interpreter is distributed throughout every one of the service routines.

Arguments to a service routine can also be placed in-line following the routine address. The routine picks up the arguments using the pointer register, each time advancing the pointer for the next use. For this, both the auto-increment and deferred auto-increment addressing modes are ideal.

For example, the following service routine copies onto the stack the value of an integer variable whose address follows the call:

```
$PUSHV: MOV @(R4)+, -(SP)
        JMP @(R4)+
```

Similarly, the following routine pops a value from the stack and stores it in the variable whose address follows the call:

```
$POPV:  MOV (SP)+,@(R4)+
        JMP @(R4)+
```

Using the two primitives $PUSHV and $POPV, the FORTRAN assignment statement:

```
I = J
```

can be implemented by "compiling" code as follows:*

```
$PUSHV  ; Address of $PUSHV routine
J       ; Address of storage for J
$POPV   ; Address of $POPV routine
I       ; Address of storage for I
```

The principal disadvantage of a normal interpreter is avoided by representing the address of a service routine in symbolic fashion as the name of a module to be obtained from a library of routines. Only those routines that are actually referred to are included in the program when it is linked for execution.

We complete this introduction by briefly illustrating how flow of control and changing modes is accomplished.

A simple transfer of control, e.g., the FORTRAN statement:

```
GOTO 100
```

can be compiled to:

```
$GOTO,.100
```

using the service routine:

```
$GOTO:  MOV   (R4),R4
        JMP   @(R4)+
```

The implementation of the FORTRAN-computed GOTO statement is illustrated in Figure 1. Notice that the count of the number of labels is included in the arguments to the service routine. The service routine checks that the index value is in the correct range; if it is not, an error is reported and control continues in-line (no transfer takes place). In this example, register 1 (R1) is used as a temporary location within the service routine.

To enter threaded code mode when executing normal code, the following call is executed:

```
JSR R4,$POLSH
```

*In subsequent examples, the arguments of a service routine will be written on the same line as the routine address. Thus, the above would appear as:

```
$PUSHV,J
$POPV,I
```

This is more compact and suggestive of conventional assembler notations; the effect is identical to the previous example.

```
FORTRAN SOURCE

            GOTO   (100,200,300) I
100         . . .
200         . . .
300         . . .

THREADED CODE

            $CGOTO,I,3,.100,.200,.300
.100:       . . .
.200:       . . .
.300:       . . .

COMPUTED GOTO SERVICE ROUTINE

$CGOTO:     MOV    @(R4)+,R1       ; Fetch value of index
            BLE    1$              ; Error if less or equal zero
            CMP    R1,(R4)         ; Compare with label count
            BGT    1$              ; Error if greater
            ASL    R1              ; •2 for word offset
            ADD    R1,R4           ; Pointer to target label
            MOV    (R4),R4         ; Fetch target label
            JMP    @(R4)+          ; Continue . . .

1$:         ERROR  "Computed GOTO value out of bounds"
            MOV    (R4)+,R1        ; Fetch label count, adjust R4
            ASL    R1              ; •2 for word offset
            ADD    R1,R4           ; Pointer to next in line
            JMP    @(R4)+          ; Continue . . .
```

Figure 1. Threaded code for FORTRAN-computed GOTO statement.

Threaded mode begins immediately following this call. The service routine is:

```
$POLSH: MOV   (SP)+,R4
        JMP   @(R4)+
```

Leaving threaded mode requires no service routine at all; the operator is simply the address of the immediately following word of memory.

A Virtual Machine

By now it should be apparent that we have the beginning of a FORTRAN *virtual machine.* Instructions in this machine language are encoded as the addresses of the service routines. The PDP-11 instruction set provides the pseudo-microinstruction set used to emulate the FORTRAN machine. Register 4 (R4) is the virtual program counter.

For a complete characterization of a virtual machine, it is necessary to identify the complete state of the machine, that is, all of the values that must be preserved in order to interrupt the execution of the machine, apply the machine to another purpose, and later resume the original execution as though the interruption had not occurred. In this sense, the state clearly includes the stack pointer (SP) register and the program counter (R4) register as well as the memory regions occupied by the program, variables, and values on the stack. In the actual implementation, some virtual machine instructions also left values in general register 0 (R0) or in the processor condition codes for use by the subsequent virtual machine instruction. Thus, these values must also be considered part of the virtual machine state. However, the remaining general registers of the PDP-11 are not part of the state even though they are used freely by individual instructions to hold temporary values during the execution of a single virtual instruction, as illustrated in Figure 1.

This FORTRAN machine went through two phases of development. In the first phase, the virtual machine specification did not change; rather, the implementation was broadened to take advantage of newer models of the PDP-11 family. Increased performance was achieved through improved performance of the new CPU and the floating-point hardware options. In the second phase, the virtual machine specification itself was extended to achieve greater performance across all of the PDP-11 family processors.

FORTRAN MACHINE – PHASE 1

The introduction described the basic technique, threaded code, by which it was possible to produce a FORTRAN processor for the first PDP-11 processor, the PDP-11/20. This section focuses on the design of the FORTRAN virtual machine proper and how it was implemented across the range of PDP-11 CPUs.

The major part of the FORTRAN virtual machine was relatively *ad hoc* in form, more or less closely following the form of the FORTRAN language. The previous example of the computed GOTO statement is representative of the approaches taken. This correspondence between the language and the virtual machine greatly simplified the compiler. Variations in the order of arguments and/or the introduction of extra arguments (such as the label list count) were made to aid the speed and/or the error checking capability of the supporting service routines.

One part of the machine had a more regular structure – assignment statements and expression evaluation. We will focus our attention on this part of the machine because this is where the majority of FORTRAN execution time is spent.

Many details of the machine are easily sketched. It was a stack-oriented machine – values were pushed onto the stack, and operators took their operands from the stack and replaced them with the result. The hardware stack pointer (SP) was used to control the value stack. Consideration was given to using the PDP-11 general registers as fast top-of-stack locations. However, this was rejected because it violated the inherent simplicity of the pure stack model and because analysis showed that the extra overhead of managing these locations substantially eliminated any benefits.

Naming conventions were adopted for the operators as a mnemonic convenience. The arithmetic operators were named as illustrated in Figure 2. For example, $ADR designated the routine to add two single-precision (real) operands, while $ADC designated the routine to add two complex operands, and so on.

Throughout this design process the size of the generated code continued to be the most important factor. This led to the most unusual aspect of the machine design.

To push a value onto the stack required two words: one for the push instruction and one for

FORM: $sot

WHERE o	=	AD	For addition
	=	SB	For subtraction
	=	ML	For multiplication
	=	DV	For division
	=	PW	For exponentiation (raising to a power)
t	=	B	For byte data
	=	L	For logical data
	=	I	For integer data
	=	R	For real data
	=	D	For double-precision data
	=	C	For complex data

NOTE:
"$PW" has a 2-letter suffix. The first indicates the base data-type, the second the exponent data-type.

Figure 2. FORTRAN Phase 1 arithmetic instructions.

the address of the variable. To reduce this to a single word, the compiler produced a service routine for each variable that would push the value of the variable onto the stack. Such a routine was called a push routine. In this way, the compiler reduced the size of the compiled code by producing specialized service routines that complemented the general service routines obtained from the FORTRAN library.

For example, the push routine for an integer variable, *I*, would be:

```
$P.I:     MOV   I,-(SP)
          JMP   @(R4)+
```

The push routine for a complex variable, *C*, would be:*

```
$P.C:     MOV   #C+8,R0
          MOV   -(R0),-(SP)
          MOV   -(R0),-(SP)
          MOV   -(R0),-(SP)
          MOV   -(R0),-(SP)
          JMP   @(R$)+
```

Of course, each push routine itself took space: three words for an integer variable and five words for a real variable. Consequently, the

*Note that since the stack of the PDP-11 grows downward in memory, values must be copied from high address toward low address to obtain a correct copy on the stack.

breakeven point was three uses for an integer variable and five uses for a real variable.

Three uses of an integer variable were deemed likely to be achieved in most programs, especially in larger and more complex programs where space would be most critical. The five uses for a real variable were reduced by some complex merging of code for multiple push routines for real, complex, and double-precision variables. The compiler also maintained a bit in the symbol table entry for each variable indicating that a push routine was actually needed. (It is fairly common for a particular subroutine to reference only a few variables out of a large COMMON block.)

Pop routines for each variable were also considered, but rejected. There are typically more uses of a variable's value than assignments of new values. Consequently, the breakeven point is less likely to be consistently achieved. Instead, general pop routines for each data-type (actually, each size of data value - 1, 2, 4, or 8 bytes) were used.

Figure 3 presents a complete example of the compiled code produced by the compiler for two sample assignment statements. The figure includes push routines automatically generated by the compiler, as well as the allocation of storage for the variables of the program. All service routines not shown are obtained from the FORTRAN library when the program is linked for execution.

It should be apparent from this figure that the compiled code corresponds to the well-known Polish postfix notation, which is a rearrangement of expression information suitable for stack evaluation disciplines.

The Virtual Machine Across the PDP-11 Family

Even as the FORTRAN system was in its early development phase, new models of the PDP-11 family were under development by the hardware groups. The next in line was the PDP-11/45 with a floating-point hardware option. How could the software development group that had just produced a FORTRAN tailored for an 8 K PDP-11/20 without even integer multiply/divide instructions respond with another FORTRAN for the high-performance

```
FORTRAN SOURCE

     K = K + 1
    X2 = (A - (B••2-4.•A•C))/(2.•A)
   END

THREADED CODE

$START:     JSR R4,$POLSH
            $P.K                        ; Push K
            $P.1                        ; Push 1
            $ADI                        ; Add integer giving K + 1
            $POPI,K                     ; Pop to K

            $P.A                        ; Push A
            $P.B                        ; Push B
            $P.2                        ; Push 2
            $PWRI                       ; B••2
            $P.4.                       ; Push 4.
            $P.A                        ; Push A
            $MLR                        ; 4.•A
            $P.C                        ; Push C
            $MLR                        ; 4.•A•C
            $SBR                        ; B••2-4.•A•C
            $SBR                        ; (A-(B••2-4.•A•C))
            $P.2.                       ; Push 2.
            $P.A                        ; Push A
            $MLR                        ; 2.•A
            $DVR                        ; ( . . . )/(2.•A)
            $POP2,X2                    ; Pop to X2

; PUSH ROUTINES

$P.K:       MOV      K,-(SP)
            JMP      @(R4)+
$P.1:       MOV      #1,-(SP)
            JMP      @(R4)+
$P.A:       MOV      #A+4,R0
            BR       $F
$P.B:       MOV      #B+4,R0
            BR       $F
$P.2:       MOV      #2,-(SP)
            JMP      @(R4)+
$P.4.:      MOV      #$R.4.,R0
            BR       $F
$P.C:       MOV      #C+4,R0
            BR       $F
$P.2.:      MOV      #$R.2+4,R0
$F:         MOV      -(R0),-(SP)        ; Shared code for pushing
            MOV      -(R0),-(SP)        ; the values of A, B, C and
            JMP      @(R4)+             ; the constants 2. and 4.

; STORAGE ALLOCATION

K:          .BLKW    1
A:          .BLKW    2
B:          .BLKW    2
$R.4.:      .FLT2    4.
C:          .BLKW    2
$R.2.:      .FLT2    2.

            .END     $START
```

Figure 3. Example of code generation.

PDP-11/45 with optional hardware floating point? Fortunately, the virtual FORTRAN machine approach made it relatively easy. All that was needed was to re-implement the virtual machine using the new and more extensive "microcode." The compiler did not even have to be changed at all! How this was accomplished is discussed below.

The PDP-11/20, with its EAE option, required two implementations of the virtual machine. The PDP-11/45 added two more: one for the floating-point option and another because it added instructions for integer multiply/divide and multiple bit shifting as part of the standard instruction set.*

Later the PDP-11/40 added a fifth variation for its Floating Instruction Set (FIS) option.†

By the time we were done, there were five versions of the FORTRAN machine which corresponded to the family processors as follows:

1. Basic PDP-11/20, PDP-11/40
2. EAE PDP-11/20 with EAE, PDP-11/40 with EAE

   Integer multiply/divide
3. EIS PDP-11/40 with EIS, PDP-11/45

   Integer multiply/divide
4. FIS PDP-11/40 with EIS and FIS

   Integer multiply/divide and single-precision floating point
5. FP11 PDP-11/45 with FP11

   Integer multiply/divide and single/double precision floating point

Later processors (PDP-11/70, 11/60, 11/34, 11/05, 11/04, and LSI-11) have all matched one of these five categories.

Figure 4 illustrates the general logical structure of a typical floating-point service routine. As presented in this logically extreme form, it consisted of *five* completely independent implementations. They were combined in a single source file to help manage and minimize the proliferation of files. (This also significantly

```
$ADR:     .IF NDF EAE!EIS!FIS!FPP
          <<no option basic implementation>>
          .ENDC

          .IF DF EAE
          <<EAE version>>
          .ENDC

          .IF DF EIS
          <<EIS version>>
          .ENDC

          .IF DF FIS
          <<FIS version>>
          .ENDC

          .IF DF FPP
          <<FPP version>>
          .ENDC

          .END
```

NOTE:
In the PDP-11 MACRO assembler language, ."IF" introduces a sequence of statements (instructions) that are included in a given assembly only if a specified condition is satisfied. The statement, ".ENDC" terminates the sequence. Also, conditional sequences can be tested within other conditional sequences, as illustrated in other figures. In this figure, the condition, "DF EAE" is satisfied if the name EAE has a defined value. "DF EIS" is satisfied if EIS is defined, and so on. The condition, "NDF EAE!EIS!..." is satisfied if none of the given names has a defined value.

Figure 4. General logical structure of conditionalized FORTRAN operator routine.

*These Extended Instruction Set (EIS) operations were similar in function to the capability of the EAE, but were an integral part of the instruction set instead of an I/O bus add-on. This was more efficient since the initialization necessary to begin execution of these functions was less.

†On the PDP-11/40, the EIS instructions were an option also.

aided maintenance.) This one file would be assembled five times, each time with a different conditional assembly parameter, to produce the five different object files that implemented the same operation on the different systems.

In practice, the separation of implementations was not as complete as shown. Some instructions, such as the computed GOTO, remained independent of the hardware configuration. Generally, the EIS and EAE versions were localized variations of the basic (no option) implementation, while the FP11 and FIS versions tended to be totally distinct.

A more representative illustration of the kind of conditionalization used is shown in Figure 5. Notice that the conditional use of EIS or EAE operations is nested within an outer conditionalization for neither FIS nor FP11. The FIS and FP11 versions are distinct.

```
$ADR:   .IF NDF FIS!FPP
        <<basic implementation>>

            .IF DF EAE
            <<EAE variation>>
            .ENDC

            .IF DF EIS
            <<EIS variation>>
            .ENDC

            .IF NDF EIS!EAE
            <<no option variation>>
            .ENDC

        <<basic implementation>>
        .ENDC ;NDF FIS!FPP

        .IF DF FIS
        FADD SP
        JMP @(R4)+
        .ENDC ;DF FIS

        .IF DF FPP
        SETF
        LDF     (SP)+,F0
        ADDF    (SP)+,F0
        STF     F0,-(SP)
        JMP     @(R4)+
        .ENDC   ;DF FPP

        .END
```

Figure 5. Partial detail of implementation of $ADR.

The FORTRAN Machine and the PDP-11/40 EIS

Because of the incompatibility in operand addressing capability between the FP11 and FIS, the FIS option of the PDP-11/40 seems at best an architectural curiosity and at worst an unfathomable aberration. In a broader perspective, however, it was an excellent compromise between goals and constraints for the combined hardware and software system at the time it was introduced.

The marketing requirement was simple. There must be at least a single-precision floating-point option for the PDP-11/40 to maintain competitive FORTRAN performance and it must sell for no more than a given (relatively low) price. The cost constraint, combined with other engineering factors, precluded the implementation of even a simple subset of the FP11 instruction set.

Consultation between the hardware and software engineers led to the resulting Floating Instruction Set. The FIS provided four single-precision floating-point instructions (add, subtract, multiply, and divide) which corresponded exactly with the FORTRAN virtual machine requirements. As seen in Figure 5, the FIS version of the FORTRAN $ADR service routine consists of just two single-word instructions (compared to the FP11 variant that occupies five words).

The FIS option for the PDP-11/40 accomplished everything that it was supposed to accomplish.

FORTRAN MACHINE – PHASE 2

While the FORTRAN product successfully "supported" the full range of the PDP-11 family, the design tradeoffs made for the original and low end of the family were not valid at the high end. Benchmark competition of FORTRAN on the PDP-11/45 with FP11 became significant even though the underlying hardware was the fastest available by clear margins. The reason is easy to understand. The FORTRAN virtual machine and its implementation did not fully exploit the hardware capability.

To illustrate, consider the execution of the statement, $I = I + 1$, as shown in Figure 3. This statement compiled to five words of threaded code (not counting the overhead of service or push routines), and required 18 memory cycles to execute. In conrast, the single PDP-11 instruction, INC I, would obtain the same effect with only two words of code and three memory cycles to execute. Similar overheads existed for floating-point operations. As shown in Figure 5, the basic arithmetic operators had to copy their operands from the stack into the FP11 registers to do the operation, and then immediately return the result to the stack.

On the PDP-11/20, integer execution times of 20 microseconds instead of 4 microseconds did not matter much when floating-point times where typically 300 to 1000 microseconds. However, with FP11 times under 10 microseconds for these operations, the tradeoffs are much different.

Since the existing compiler was based totally on the threaded code implementation, a complete new compiler that generated direct PDP-11 code would be needed to fully exploit the hardware potential. In the meantime, something was needed to immediately improve performance and relieve the competitive pressure.

That something was provided, not by discarding threaded code, but by extending the FORTRAN virtual machine architecture. The extension devised was based on a combination of systematic and *ad hoc* pragmatic considerations.

The primary considerations were to:

1. Focus attention on operations for integer, real, and double-precision data-types. Logical and complex data-types do not occur frequently enough to merit much concern [Knuth, 1971].
2. Limit the impact on the compiler to as small a portion as possible to limit the programming effort. Fortunately, expression handling and assignment statements were well modularized in the implementation.

Addressing Modes

The principal concept that formed the basis of the extended machine was the recognition that operands could be in any of a number of locations and that arithmetic operators should be able to take operands from any of them and deliver the result to any of them, instead of just the stack. The principal locations identified were:

- The stack.
- In memory at an address given as a parameter.
- In memory at an address given in R0 as a result of an array subscripting operation.

Other "locations" were formalized for particular groups of operators as will be seen later.

Conceptually, these locations became addressing modes associated with each operator. However, any kind of decoding of addressing modes during execution would destroy the performance objective. Consequently, each combination of operator and addressing modes was implemented by a unique threaded service routine.

At this point, a new consideration came into play. Not only would each routine take some memory, but the number of global symbols that must be handled by the linking loader would rise dramatically. (The system linking loader maintained its global symbol table in free main memory; hence, the number of symbols that could be handled was limited by main memory size. Fortunately, the minimum system main memory requirement had independently increased from 8 Kwords to 12 Kwords; otherwise, the approach would not have been acceptable.) The above three modes for each of three operand locations for each of the four basic operations for each of the three important data-types required 3 * 3 * 3 * 4 * 3 or 324 new service routines. Care would be needed to keep this explosive cross-product in bounds.

The memory size increase was offset by the fact that in many cases the push routines of a variable were no longer needed. This can be appreciated better by looking at some examples.

The Extended Machine

Figures 6 through 11 detail most of the extended machine and give numerous sample code sequences.

There were three principal groups of extended operations dealing with one-dimensional array subscript calculation, arithmetic operations, and general data movement. Once again, naming conventions were used for mnemonic aids. Generally, the first two or three letters (after the "$") designated an addressing mode, the next letter designated the kind of operation and the final letter designated the data-type. For example, the $ADR routine used in previous figures acquired the name $SSSAR in this new scheme.

As an example, consider the FORTRAN statement:

```
I = J + K + L
```

This would be compiled to:

```
$CCSAI,J,K      ; Add J,K and
                ; put result on stack
$SCCAI,L,I      ; Add stack,L and
                ; put result in I
```

The PDP-11 code for these service routines is:

```
$CCSAI:   MOV   @(R4)+,-(SP)
          ADD   @(R4)+,@SP
          JMP   @(R4)+

$SCCAI:   ADD   @(R4)+,@SP
          MOV   (SP)+,@(R4)+
          JMP   @(R4)+
```

```
FORM: $sbXz, sarg, barg

WHERE        s  =  C        If subscript is in memory (core) and directly
                            addressable (i.e., not a parameter or array element)
                =  R        If subscript is pointed at by R0 at execution time
                =  S        If subscript on execution stack
                =  P        If subscript is a parameter
                =  G        If subscript is contents of R0 (i.e., results of
                            function call)
             b  =  C        If array is not a parameter
                =  A        If array is a parameter
             z  =  1,2,4,8  The array element size in bytes
          sarg  =  Argument address if s = C
                =  Argument list offset if s = P
                =  Not present otherwise
          barg  =  Array address minus element size if b = C
                =  Address of array descriptor block (ADB) if b = A

SPECIAL CASES

      $CCX0, address

            Is generated when the subscript is a constant and the array is not
            a FORTRAN dummy argument. The final address is computed at compile
            time and is the argument.

      $KAX0, scaled-constant, adb-address

            is generated when the subscript is a constant and the array is a
            FORTRAN dummy argument; the constant subscript is converted to a
            byte offset at compile time.
```

Figure 6. One-dimensional array subscripting instructions.

ASSUME

```
SUBROUTINE SUB(A,I)
DIMENSION A(10), B(10), M(10)
```

FORTRAN SOURCE	COMPILED CODE
B(J)	$CCX4,J,B–4
B(I)	$PCX4,4,B–4
B(5)	$CCX0,B+20
A(5)	$KAX0,20,$A.A
B(M(2))	$CCX0,M+2 $RCX4,B–4

NOTE:
$A.A is the address of an array descriptor block for A.

Figure 7. Example of subscripting operations.

FORM: $1rdot, larg, rarg, darg

Where	1	= C	If argument is in memory (core) and directly addressable (i.e., not a parameter or array element)
		= R	If argument is pointed to by R0 at execution time (i.e., as the result of a subscripting operation)
		= S	If argument is contained on the execution stack (SP)
		= D	If D (destination) is C and is the same argument
	r	= C	(As above)
		= R	(As above)
		= S	(As above)
		= K	If argument is in core, directly addressable, and an integer constant (i.e., special case of C)
		= 1	If argument is integer constant 1 (i.e., special case of K)
	d	= C	(As above)
		= R	(As above)
		= S	If result is to be placed on execution stack
	o	= A	For addition
		= S	For subtraction
		= M	For multiplication
		= D	For division
	t	= I	For integer data
		= R	For real data
		= D	For double-precision data

larg, rarg, darg
= Argument address if addressing mode = C
= Constant value if addressing mode = K
= Not present otherwise

Figure 8. Arithmetic instructions.

Move instructions are two address instructions. Data of any type may be moved.

FORM: $sdVt, sarg, darg

Where	s	= C	If argument is in memory (core) and directly addressable
		= R	If argument address in R0 at execution time
		= S	If argument on stack
		= G	If argument contained in R0–R3 (as result of function call)
		= K	If argument is integer constant
		= 1	If argument is integer constant 1
	d	= C	(As above)
		= R	(As above)
	t	= B	For byte data
		= L	For logical data
		= I	For integer data
		= R	For real data
		= D	For double-precision data
		= C	For complex data

sarg, darg = Argument address if address mode = C
= Constant value if address mode = K
= Not present otherwise

Figure 10 Move instructions.

ASSUME

DIMENSION L(10)

FORTRAN SOURCE	COMPILED CODE
A = B + C	$CCCAR,B,C,A
A = B + C·D	$CCSMR,C,D $CSCAR,B,A
I = J + 5	$CKCAI,J,5,I
I = I - 5	$DKCSI,5,I
J = J + 1	$D1CAI,J
L(J + 1) = J + 2	$C1SAI,J $SCX2,L-2 $CKRAI,J,2
I = L(I) + 2	$CCX2,I,L-2 $RKCAI,2,I

Figure 9. Example of arithmetic operations.

ASSUME

DIMENSION ARRAY (10)

FORTRAN SOURCE	COMPILED CODE
A = B	$CCVR,B,A
I = 1	$1CVI,I
B = ARRAY(J)	$CCX4,J,ARRAY–4 $RCVR,B
ARRAY(1) = ARRAY(I+1)	$C1SAI,I $SCX4,ARRAY–4 $GET3 $CCX0,ARRAY+0 $SRVR

Figure 11. Example of move instructions.

Notice that no push routines are needed for any of the variables.

All subscripting operations resulted in the address of the array element being left in R0 at execution time. Only one-dimensional arrays were handled. Two- and three-dimensional arrays continued to be handled as in the more general Phase 1 implementation.

These forms can occur on both left- and right-handed sides of assignment statements.

The arithmetic instructions are three address instructions, taking two arguments and putting the result in a designated place. These instructions are limited to +, –, *, / on integer, real, and double-precision data.

Ad Hoc Special Cases

Within this general framework, a number of additional *ad hoc* addressing modes were incorporated.

For each of the arithmetic operators and each of the three data-types, the first operand addressing mode could be given as D to designate that it was the same as the destination core address and the destination parameter was eliminated. This was not done for the second operand based on the simple observation that programmers will almost always write assignments as:

```
A = A + . . .
```

instead of:

```
A = . . . + A
```

This added 12 more service routines.

For the integer operators only, the second operand could be given as K to designate that it was a constant given as the parameter instead of the address of the value. This was not done for the first operand for reasons similar to the case above.

For integer add and subtract operators only, the second operand could be given as 1 to designate that it is the constant value 1 and no parameter is present. This is simply a frequent special case of the previous use of K.

By combining the above, the FORTRAN statement:

```
K = K + 1
```

is compiled to:

```
$D1CAI,K
```

where the service routine is simply:

```
$D1CAI:    INC    @(R4)+
           JMP    @(R4)+
```

This code occupies two words and requires five memory cycles to execute. This is not quite as good as the two words and three cycles needed for direct PDP-11 code, but far better than the five words and 18 cycles required by the earlier implementation.

General Results

Execution improvement varied, of course, with the particular programs used. Over a large set of programs, the following guidelines were obtained.

- Programs that were floating-point intensive increased in speed by factors of 1.1 to 1.6, with 1.3 being representative.
- Programs that were integer intensive increased in speed by factors of 1.4 to 2.4, with 2.0 being representative. (One particularly simple benchmark increased in speed by a factor of 4!)

Moreover, because of the reduced need for push routines, most programs increased in size by less than 10 percent.

The improvement for integer operations was better than for floating-point operations for several reasons. Integer operations were more easily "optimized" because they took place in the basic CPU general registers. The FP11 has a separate set of floating-point registers, and floating-point computations must be performed only in those registers. Also, the FP11 operates in either single-precision or double-precision mode depending on a status bit; the compiler implementation was not suitable for tracking the state of this bit and, hence, each floating-point operation continued to bear the overhead of reestablishing the state as needed by that operation. (This is the purpose of the SETF instruction shown in Figure 5.)

The performance improvements of the Phase 2 system with its extended virtual machine were obtained with a design, development, and testing effort of about three man-months. For that effort, PDP-11 FORTRAN regained a strong competitive position that held reasonably well until FORTRAN IV-PLUS, an optimizing PDP-11 code-generating system, replaced it 18 months later (in early 1975).

REAL MICROCODE AND THE FORTRAN MACHINE

Clearly, the FORTRAN virtual machine described above could be implemented in "real" microcode instead of the PDP-11 instruction set. This was considered during the design planning for the PDP-11/60 which features a writable control store microprogramming option [DEC, 1977a]. But, while the analysis showed that a significant improvement could be obtained, the result, at best, would be comparable to the performance already achieved by the FORTRAN IV-PLUS product. Consequently, it was not done.

The analysis proceeded along the following lines. Execution time was considered in three categories: instruction fetch and decode, operand fetch and/or store, and execution time proper. Since the analysis is a comparison of different FORTRAN implementations for a given machine, the basic execution times are assumed to be the same and neglected. The resulting comparison, thus, shows the number of words of memory and the number of memory cycles for each implementation.

For this presentation we shall consider the following two FORTRAN statements as reasonably representative of FORTRAN as a whole.

```
    I = J * K + L
 A(I) = B(J) + 4
```

For these statements, the size and memory cycles are easily determined by examination of the code generated by FORTRAN and FORTRAN IV-PLUS, respectively. These values are shown in Table 1.

For the hypothesized micro-thread implementation, the code size is unchanged from FORTRAN, while the memory cycle count is

Table 1. Comparison of Size and Time Requirements of Sample Statements with Different Implementation Techniques

	I = J * K + L		A(I) = B(J) + 4	
Technique	**Size**	**Time**	**Size**	**Time**
PDP-11 threads	6 words	20 cycles	9 words	38 cycles
FORTRAN IV-PLUS	8 words	12 cycles	14 words	20 cycles
Micro-threads	6 words	12 cycles	9 words	22 cycles
Model	7 words	11 cycles	9 words	17 cycles

reduced by eliminating the instruction fetches that occur in the service routines. These results are also shown in the table. Comparison of the results shows that the micro-thread implementation is faster (as expected), but also that its speed is no better than that of FORTRAN IV-PLUS. Could this be coincidence or is there reason to believe these results should be obtained?

To answer this, we formulated a simple intuitive model for the expected size and speed of code on an idealized FORTRAN machine. To estimate the code size:

- Count one unit for each variable that is referenced (e.g., A(I) counts as two).
- Count one unit for each operation performed (e.g., assignment or subscripting are unit operations).

To estimate the memory cycles for execution:

- Count one unit for each variable that is referenced.
- Count one unit for each operation performed.
- Count one, two, or four units for each value fetch or store operation depending on the size of the data.

This very simple model is appropriate only for compilers that produce code based only on isolated source information, which is true of the original FORTRAN. Optimizing compilers, such as FORTRAN IV-PLUS, do better than suggested by this model by eliminating or simplifying operations (for example, by constant expression elimination or moving invariant computations out of loops, and/or by keeping values in registers instead of main memory, especially across loops). Consequently, the model serves primarily as a relatively implementation-independent frame of reference for comparing alternative implementations.

The sizes and cycle counts from this model for the sample statements are also shown in Table 1. These values are quite similar to values for both the micro-thread and FORTRAN IV-PLUS implementations.

We interpreted these results as a clear demonstration that a micro-threaded implementation could not significantly outperform the existing FORTRAN IV-PLUS implementation. Further, effort expended for greater performance would be better directed toward improved optimization in FORTRAN IV-PLUS (which would benefit existing hardware products) or toward faster hardware *per se.**

There is also a broader interpretation of the results that is worth reflection. The threaded implementation was designed to be a good FORTRAN architecture. Yet, when implemented in microcode in a manner comparable with the host PDP-11 architecture, the performance is close to that achieved by the FORTRAN IV-PLUS compiler and also close to that of an "ideal" model. One is led to speculate that the PDP-11 with FP11 is also a good FORTRAN architecture.

ACKNOWLEDGEMENTS

Many individuals contributed to the design, implementation, and evolution of the PDP-11 FORTRAN product. The following were particularly involved in those aspects described in this paper. Jim Bell, Dave Knight, and the author participated in the initial design evaluation that led to the basic virtual machine. Dave was project leader for the first versions of the product. Rich Grove participated in the support of the FP11 and FIS options. The extended virtual machine design and implementation, and the microcode feasibility analysis were done by the author. Finally, Craig Mudge assisted in the preparation of this paper with valuable discussion and criticism, and by not accepting "no" for an answer.

*Note that Digital did both. FORTRAN IV-PLUS V2 and the FP11-C were both released in early 1976 with each offering significant performance improvements.