SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 601–616 (1978)

Experience with the Universal Intermediate Language Janus

B. K. HADDON AND W. M. WAITE

Department of Electrical Engineering, University of Colorado, Boulder, Colorado 80309, U.S.A.

SUMMARY

Janus is a symbolic language intended for use as an intermediate language in the transportation of software. Since its initial design four years ago, it has been used to implement a portable Pascal compiler, in the design of an Algol 68 compiler, and to realize a portable package of mathematical routines. These experiences, together with a critical re-evaluation of the design criteria, have led to some modification of the specifications of Janus and an increased confidence in the viability of the approach. They have also indicated some problems yet to be solved. This paper reviews the significant lessons which we have learned, and quotes some results which support our confidence.

KEY WORDS Portability Intermediate language UNCOL Janus Abstract machine model

INTRODUCTION

Janus is a symbolic language used to embody the information normally passed from the analysis phase of a compiler to the code generator.[1] Its main purpose is to serve as an intermediate language to enhance the portability of software written in high-level languages.[2] It has also been successfully used by human programmers for coding low-level support routines.

Our conceptual orientation is that the Janus language is the 'assembly' language for the Janus machine, and the structure of the language reflects explicitly the structure of the modelled machine. The reasons for depending so strongly upon an abstract machine model are explained elsewhere,[3] but one of them should be reiterated here: By thinking in terms of a machine, we provide ourselves with an Occam's razor when considering extensions of or modifications to the language. In attempting to accommodate the intermediate language to some higher-level feature, we look for a realization in terms of a machine, and then give it syntactic and semantic form.

We will adopt the following convention for the remainder of this paper: the word 'compiler' shall be used when speaking of the language processor that accepts a high-level source program and produces the intermediate Janus text, and the word 'translator' shall be used when the processor that generates target code from the Janus is intended.

In the next section we shall briefly trace the history of Janus, placing the projects from which we have drawn our experience into their proper context. The following section summarizes some of the changes made to the Janus abstract machine model as a result of the experience and insight gained since its initial design. Since the most detailed exposition

0038–6644/78/0508–0601$01.00

Received 16 January 1978
Revised 3 May 1978

of the original design was given by Coleman,[2] we shall refer to it as the 'Coleman machine' (although imputing no sole responsibility).

Our attempts to develop a suitable model for temporary storage are presented in some detail in the third section. The problems discussed there are indicative of a theme which recurs throughout the paper: in order to preserve information for the translator from the compilation phase the Janus code must reflect as far as possible *what* is happening, and avoid attempting to dictate *how* to accomplish it. One of the most important lessons to be learned from our experience is the central role played by this principle.

Several significant problems remain unresolved; we indicate their dimensions but do not attempt to speculate on their solution. In spite of these difficulties, we believe Janus to be a viable tool and present some results in support of this belief.

At the appropriate points we shall review the particular aspects of Janus necessary to understand our arguments; we assume, however, that the reader is thoroughly familiar with the general concepts of programming language design and implementation.[4]

HISTORY

Janus was first described by Coleman *et al.*,[1] and the abstract machine model that is represented by the Janus language was later further refined by Coleman.[2] The initial work relied heavily upon earlier experience with the use of abstract machine models to implement portable software,[3] which indicated that the models employed had many features in common. Thus the possibility of a universal model incorporating these common features was examined, and resulted in the Janus abstract machine. This model has evolved as we have gained further experience, leading to the present definition.[5]

The Janus abstraction

The design of Janus is based upon the observation that the translation of a programming language can be divided into two tasks: the analysis of the source text to express the program in terms of the abstractions of the given programming language, and the translation from the abstractions of the programming language to the abstractions of the target machine.

One directing principle in the design of the Janus language is that it be a symbolic representation of the interface between these tasks, and incorporate in its structure and instruction repertoire those abstractions found in many programming languages. The suitability of Janus for the representation of algorithmic processes is in part a measure of the effectiveness of these abstractions in higher-level languages. Another directing principle for the design of Janus is that the re-expression of Janus in terms of target abstractions should be as inexpensive (both intellectually and algorithmically) as is feasible. This implies that no concept in Janus should be too far away from one which we know can be implemented with relative convenience on current hardware. It is also important that we are able to analyse the syntactic form of Janus with tools at our command (such as the STAGE2 macro processor[6,7]). As will become apparent in the subsequent discussion, achievement of just the right balance between these two principles has been, and continues to be, a task requiring attention to both broad generalizations and individual details.

The PascalJ compiler

The PascalJ compiler project was an attempt within our group to construct a fully self-porting ('bootstrappable') system implementing the Pascal language.[8] The aim of this project was to write the compiler in Pascal, to compile the compiler using the Pascal system available to us, then to use the resulting compiled program to compile its own source, hence obtaining a Janus version of the compiler. The full bootstrap could then be accomplished by implementing Janus on the target machine. These aims were in fact realized: the PascalJ compiler was developed on a CDC6400, and then transported to a Sigma 3 computer here and to an Interdata machine at the University of Melbourne. (This latter implementation was carried out by a person not a member of our group.[9]) The development of this compiler refined our notions concerning many details of Janus, particularly those associated with the control of the lifetimes (extent) of temporary objects (see the section entitled *Allocation of Temporaries*). However, the most important lessons were those associated with the effort needed to perform the full bootstrap and with the engineering of the compiler itself.

In order to transport the compiler, we found it necessary to implement Janus (via STAGE2) in full, as we had placed no limitations on the facilities of Janus used within the compiler. Experience led to the estimate that at least six man-months would be required to define the Janus machine in terms of a target machine, assuming reasonable familiarity with both on the part of the implementor. This situation has been improved considerably by the development of lower-level abstractions which ease implementation at the cost of restricting the class of target computers (see discussion in the section entitled *Some Results*).

This initial version of the PascalJ compiler was constructed as a monolithic one-pass processor. This presented no problems when executing on a sufficiently large machine, apart from the cost of compiling the system using the available Pascal compiler. However, the PascalJ compiler was only able to execute on smaller machines by manipulating the run-time image to permit considerable overlaying during execution. Although these observations are only incidental to the design of the Janus language, they do serve to emphasize that portability is achieved only through total system engineering; guaranteeing the portability of the intermediate language is not of itself sufficient.

Algol 68

The use of the Janus intermediate language in the design of an Algol 68 compiler[10] has further tested the validity of the abstract modelling concept. The designer has expressed some doubt about the sufficiency of the Janus machine, but states that 'the effects of choosing Janus as the intermediate code have been a considerable clarification of the object code generator'.

A source of difficulty has been his design objective of writing the run-time storage management routines in Algol 68 itself. We do not have information on precisely how this objective is to be accomplished, but it has been proposed that the compiler should be able to generate declarations in the Janus code that specifically supply information for the garbage collector. Such information is not normally available to the Algol 68 programmer,[11] hence we expect that additional operations will have to be provided to access this information at the Algol 68 level.

The Janus definition assumes the existence of a minimal storage management scheme in order to implement its GRAB and FREE operations. The implementation of Algol 68 outlined above presumes some interaction with the Janus manager, and thus requires it to understand some of the details of the Algol 68 storage organization. This we regard as being most unfortunate, as it implies that portions of a Janus implementation would have to be rewritten depending on the high-level language being modelled.

It is our opinion that the storage management requirements of high-level languages are an intimate part of each language, and thus exist at a higher level of abstraction than that of the Janus machine (as in the case of parameter passing conventions, where Janus provides only a value mechanism as this can be used to implement mechanisms via suitable code sequences). The portability of the storage manager for a particular language is then attained by writing it in Janus (see below) relying upon the GRAB and FREE primitives merely to perform the final mapping into the addressing structure of the target machine.

Machine-independent mathematical routines

As part of the total approach to the attainment of portability, a set of basic mathematical routines based on the work of Cody[12, 13] has been coded in Janus. Careful attention was paid to parameterizing the code in order to preserve the accuracy of the algorithms across a range of precisions, number systems and bases (see the section on *Some Results*).

It is important to note that these routines were hand-coded directly in Janus. At the time Janus was conceived it was not thought that humans would ever need to write Janus code, and still the primary intention is that Janus be a compiler intermediate language. We have found for these service routines, however, that hand-coding gives a more appropriate degree of control over both the Janus code and the generated target code (the classic reason for working in assembly language). Surprisingly, Janus coding has not been found to be all that difficult. We are now considering writing a more extensive run-time support system in Janus, particularly implementing the activation stack for recursive procedures.

In order to write the mathematical routines, and to be able to contemplate the recursion support, we had to refine our notions concerning non-recursive procedures (especially the specification of the non-local environment of such procedures). Problems with the denotation of real numbers were also encountered. (Both of these subjects are discussed in detail below.)

CHANGES FROM THE COLEMAN MACHINE

The gross organization of the Coleman machine has survived largely intact into the current model. The processor, memory and LIFO stack constitute major structures of the present Janus machine as they did in the Coleman machine, although each is changed in detail. The instruction format of the Coleman machine is also altered only in detail.

Accumulator and index register

The Coleman machine incorporated an accumulator and an index register as separate structures. It had been felt that the combination of an explicit accumulator and a stack surpassed the pure stack as a model for single-accumulator and multiple-register machines. In practice it was found that a compiler for the Janus machine had to perform a register allocation in order to be able to use the accumulator (and the index register), and then another register allocation scheme was required in translating the Janus to a target machine. Apart from the duplication of effort, it is not clear that an optimal strategy at the Janus level would (or even could) be translated to an optimal strategy on the target machine. The reason is that the distinction between the accumulator and the stack at the Janus level of abstraction could cause some shuffling of operands unnecessary on a multiple-register machine or pure stack machine, and the translator to the target machine could not detect this as being waste motion.

The properties of the index register in the Coleman machine effectively gave it the capabilities of a second accumulator, which proved difficult to implement if the target machine index register did not have these properties, or if a second accumulator was not available. An explicit assumption in the Coleman design was that index computations could be performed while holding a value of interest in the accumulator.

These difficulties were circumvented by making the stack an operand stack which allows operations to be performed upon the top element or elements. This removed the need for an accumulator as a separate structure, and the associated allocation problem. The inherent pushdown nature of a stack permits index computations to be performed while holding values of interest further down the stack. Thus the index register is now regarded simply as a register in the processor, having no computational ability, serving only as a possible element in the memory accessing function. The limitation on the lifetime of a value in the index register (available for use in the next instruction only) leaves the implementor free to use any indexing method that he wishes (index register, indirection, instruction modification, etc.).

Contrary to the impression given above, we have not created a 'pure' stack machine. Translation of pure stack code for a one-accumulator machine (or similar) requires contextual information to be stored so that operators can be combined with the corresponding operand addresses. This problem is discussed fully in the original design paper.[1] Hence the Janus code is 'collapsed', operations being able to specify at most one operand when desired. The expansion of this type of instruction format for a stack machine or a multiple-register machine presents no difficulties.

Memory structure

The Coleman machine provided a memory essentially linear in structure, divided into a number of categories. A reference to memory within a category consisted of a symbolic address and an index formed by summing fixed and variable offsets.

The Janus machine now incorporates a third part in a memory reference: a displacement within the structure located above. A further addition to the Janus machine is the base register, which can hold a representation of the three-part composite address. Like the index register, it is regarded as a part of the processor, taking part only in the addressing mechanism, and the value only being available for use in the next instruction.

These mechanisms give the Janus memory the tree structure of the state used in the Vienna Definition Language (VDL).[14, 15] Janus is thus able to represent any memory structure that can be described in the VDL. A memory access starts from some node in the memory structure. The symbolic address, the index and the displacement (any or all of which may be null) are treated as a named selector, a computed selector, and another named selector respectively, to arrive at a new node. By means of the base register, a further addressing sequence can be commenced from the new node. The category part of an address can be thought of as a precomputed node (whose relation to the root node of the memory is not specified). This addressing scheme handles the Pascal and Algol 68 models with little difficulty; an awkward linearization was sometimes required for the Coleman machine. This linearization forced a compiler to specify *how* a structured record was to be represented, rather than informing the translator *what* fields were present in the structure.

Amongst the categories currently defined are STATIC, whose extent is the entire program; INTERN, whose extent is one module (there may be more than one module per program and/or more than one procedure per module); and DISP, whose extent is that of the recursive procedure containing the declaration. The category field of an instruction address may also specify BASED, indicating the current node is given by the base register.

These changes have caused a small alteration in the instruction format, but it is still recognizably the original Coleman style for instructions containing references. The Coleman machine had a second format for mode conversions, but this has been abandoned entirely in favour of explicit operators performing conversions in specified ways. All in all, the concept of mode has not been found to be as important as was assumed in the design of the Coleman machine. What was the mode field of an instruction is now considered a modifier of the operator, and modifiers that are not the names of modes are permitted.

Procedure calling

The procedure call of the Coleman machine was modelled upon the commonly used method of parameter passing which stores the parameters or pointers to the parameters in a block associated with the call. Thus the calling sequence included both code to evaluate the parameters and storage allocation declarations (see Figure 1(b)). This form considered primarily the needs of non-recursive invocation, although the authors stated that they realized that an alternate form for recursive calls was required. In the latter case, the parameters cannot remain bound to the calling sequence itself—by some mechanism they must be associated with each invocation, usually by copying, or directly storing, onto an activation stack. This implies that the storage reservation be performed elsewhere, making the declarations in the Coleman form redundant. The present form of the procedure call in Janus is illustrated in Figure 1(c). By comparison of Figures 1(b) and 1(c) it can be seen that the form of a procedure call in the Coleman machine was more concerned with the how's of accomplishing the parameter and flow of control linkage than the present Janus model. The present model does, however, retain sufficient information to generate a local parameter block if this method is still to be used for non-recursive procedures.

```
F(1, A, B+C, D[I+3])
```

(a)

```
CALL REAL PROC F()
ARGIS INT CONST C1 () A 1
ARGIS,I REAL LOCAL A()
LOAD REAL LOCAL B()
ADD REAL LOCAL C()
STARG REAL TEMP T1()
LDX INT LOCAL I()
MPX INT CONST C2() E REAL
LOAD,I REAL LOCAL D(3*REAL)+
STARG ADDR ARG L1 (3*ADDR)
RJMP REAL PROC F()
SPACE ADDR ARG L1(4)+
SPACE ADDR ARG (1) A C1
SPACE ADDR ARG (1) A A
SPACE ADDR ARG (1) A T1
SPACE ADDR ARG (1)
CEND REAL PROC F(1)
```

(b)

```
CALL REAL R F01
ARG(N) INT A INT 1
ARGLOC(N) ADDR LOCAL A02
LOAD REAL LOCAL B03
ADD REAL LOCAL C04
ARG(N) REAL
LOAD INT LOCAL I05
ADD INT A INT 3
INDEX INT
ARGLOC(N) ADDR D06(0)REAL*
CEND REAL R F01
```

(c)

Figure 1. Example of a procedure call: (a) a procedure call; (b) Coleman code for (a); (c) Janus code for (a)

The more significant consideration is that of establishing the environment for the execution of the procedure. For recursive procedures, we have adopted the usual model.[16, 17]

A display level is specified for each recursive procedure—the sequence of run-time calls builds the addressing environment. A similarly convenient model for non-recursive procedures is not known to us, although it is obvious that any model must at least include the FORTRAN conventions for local storage and those FORTRAN extensions allowing multiple entry points. It was these considerations that led to the definition of INTERN category storage.

We have identified the possible necessity of a third type of procedure call to communicate with operating system supplied functions, particularly those associated with input/output. In many systems the calling conventions for these functions are distinct from subroutine library conventions. At the present time our experience in this area is limited; it has sufficed to simply flag the procedure name as the name of a system function, and to provide individual translation rules.

Division

Not all the problems that we have encountered affect the gross structure and organization of the Janus machine. Many small details must be decided using a variety of different criteria. Each demonstrates an aspect of the difficulty in steering a line between our dual objectives.

For some time, we included in the definition of Janus two integer division operators:

DIVN	$\lfloor a/b \rfloor$	Result truncated towards *N*egative infinity
DIVZ	$\text{sign}(a/b) * \lfloor \text{abs}(a)/\text{abs}(b) \rfloor$	Result truncated towards *Z*ero

We reasoned that both were necessary, as it is expensive to synthesize one from the other. The argument went that if a high-level language specified one of these operations for its integer division, e.g. DIVN, and Janus contained only DIVZ, then a compiler would have to generate Janus code to simulate the effect of DIVN. But a given target machine may in fact have only a DIVN instruction. But the translator seeing only the DIVZ in the Janus text must generate code using the DIVN of the target machine to simulate the effect of the DIVZ (the 'how' versus 'what' effect again)! However, a survey of 20 different machines found only one (the EELM KDF9[18]) that had a DIVN, and a similar survey of high-level languages found none that actually specified their integer division to be DIVN (with the reservation that a proportion of the languages did not specify the results obtained from performing integer division with operands of opposite sign—the only condition under which DIVN and DIVZ give different results). As a result, we have deleted DIVN from the operator repertoire.[19]

By a similar line of reasoning, we have two operators for real-to-integer conversion, TRUNCN and TRUNCZ performing respectively the truncations described above.[19] Contrary to the findings on division, both forms arise naturally due to the popular use of shifts for this conversion. The effect of shifting off the fractional part in a sign-modulus or one's-complement representation is TRUNCZ, in a two's-complement representation is TRUNCN. These questions have also been addressed more recently by Wirth[20] in considering divide-by-two optimizations.

It is also interesting to note the effect of the number system on the test for oddness. In sign-modulus or two's-complement it is only necessary to inspect the least significant bit of an integer (*odd* = *lsbit*). In a one's-complement representation, this property is a function of the sign and the least significant bit (*odd* = *sign* xor *lsbit*).

Real numbers

In the representation of real numbers in the Coleman version of the language the decimal point of the significand was implicitly at the right, interpreting the significand as an integer. This has proved to complicate the calculation of the exponent by the translator, since the translator is forced to continue counting digits in the significand even after they cease to be significant. We have therefore changed our conventions so that the decimal point is now assumed to be at the left, interpreting the significand as a fraction.

In developing the package of mathematical routines we have realized that the accuracy of real constants can be critical, yet that simply increasing the number of significant digits is not a solution. Apart from the potential loss of significance in the conversion from a decimal to another base,[21] the translator will frequently have significance limitations that contribute to the loss of accuracy. An example will better illustrate this problem.

In order to preserve accuracy for large argument values, one of the common range reduction techniques for sine/cosine[12] requires that a particular constant be expressed *exactly*. Consider, as an example, the following 13-bit value (expressed in octal notation):

3.1104

The representation of this value in decimal notation is

3.1416015625

Thus, 34 bits of computational accuracy are required to process the decimal representation to produce the *exact* 13-bit value. An implementor for a machine with 24 bits of floating-point accuracy may decide to scan eight, or generously nine, decimal digits of such constants, unintentionally introducing catastrophic errors.

As a consequence of the above, we have introduced real denotations with a binary base into Janus. Yet this is not a complete solution, as the compiler then has to be parameterized to produce the correct Janus form of constants depending upon the features of the target machine (the decimal form still being preferable for decimal machines). Analogous problems affecting accuracy that depend upon decimal, binary or hexadecimal normalization also exist.

ALLOCATION OF TEMPORARIES

In compiling a higher-level language, it is frequently necessary for the compiler to generate a variable to hold an intermediate result. Examples are values of subexpressions, array indexes and loop limits. These variables are anonymous to the high-level programmer (he does not have names with which to refer to them) and hence the compiler has complete control over the pattern of use.

Many temporaries, particularly those arising as intermediate results in expressions, are used in a last-in, first-out manner. The value is usually required but once, as it only enters into the computation of the remainder of the expression. The Janus operand stack models this style of use, the stack having the required LIFO property, and each operand being available at the top of the stack to be combined with others in the desired order.

Other temporaries, for example those which are the values of common subexpressions, do not follow this pattern. These are inherently used more than once, and do not follow the LIFO discipline. Manipulation of the stack (i.e. a set of operators to permute and duplicate the top few elements in several ways) was considered briefly as a solution, but was rejected because it is difficult to recover the intent of these manipulations when translating for a non-stack machine. The reason is that such a stack manipulation scheme hides from the translator that a particular value is being preserved for later use, as many other objects are moved just to accomplish the desired end. Similarly, the existence of the accumulator in the Coleman machine focused attention upon *how* to shuffle data to and from the stack rather than upon *what* operations were to be executed.

The solution adopted for temporaries not following the stack pattern is to have the compiler give them names (symbols) in TEMP category storage, thus allowing multiple reference and explicitly informing the translator of the purpose of these variables. Additional pseudo-operations are used to give the details of the variable (e.g. mode) and its final use.

Limitations on use of temporaries

With all temporary variables, the intention is that it should be possible for the translator to allocate registers (or their equivalent) as often as possible. A translation for a one accumulator machine would frequently have the accumulator represent the top stack element—a more sophisticated translation for many register machine could manage to use registers for several stack elements as well as non-stack temporaries. However, without a global analysis of flow of control, this type of optimization is only possible within a basic block (a section of code having only one entry point).

Figure 2 shows a reasonable representation of a pretested count-controlled loop, assuming the body of the loop leaves the stack in its original state. This formulation implicitly attempts to keep the counter in registers, as would a target machine assembly language programmer if he had the registers at his disposal. Unfortunately, the state of the stack cannot be determined at the label LOC L18 without looking ahead to the jump that references it. In this particular example, the stack state happens to be identical to that at the line previous to the LOC, but it is important to note this cannot be guaranteed. Note further that this sequence of code presents no problem on a real stack machine, or in an implementation that simulates a stack during execution. The problem comes from attempting to resolve the stack accesses into an equivalent sequence of register or memory accesses at translation time.

```
.
.
.
LOAD INT LOCAL G201.          COUNT FOR LOOP
JMP  R L21.                   ENTER LOOP AT BOTTOM
LOC L18.                      LABEL AT TOP OF LOOP
SUB INT A INT 1.              DECREMENT COUNT
.
.
.                             BODY OF LOOP
.
.
.
LOC L21.                      BOTTOM OF LOOP
CMP INT A INT 0.              TEST END
JMP NE R L18.                 LOOP IF NECESSARY
.
.
.
```

Figure 2. A loop that does not work

We have investigated the use of pseudo-operations at each label to specify the stack state —this can be an enormous amount of information, mostly not required, and, where a label is not anonymous (i.e. corresponds to a programmer-defined label), can require the compiler to examine the global flow of the source program. We thought for a time we had a solution using named temporaries, but there is another aspect of the problem that invalidated this approach.

The form of this proposed solution is illustrated in Figure 3. It essentially moves the contents of the stack to named temporaries (only one element of the stack is illustrated). No extra target code is implied. A STORE(N) from the stack to the named temporary variable requires that the translator note that the register (or other location) that was the top of the stack is now the named temporary, and that the top of stack is now elsewhere (or non-existent). Similarly, a LOAD from a named temporary does not necessarily require the generation of code at the target level. The point in using named temporaries is to give the translator a 'handle' on the values.

```
        LOAD INT LOCAL G201.
        TEMP INT NONREC T35.
        STORE(N) INT TEMP T35.
        JMP  R L21.
        LOC L18.
(1)     LOAD INT TEMP T35.
        SUB INT A INT 1.
        STORE (N) INT TEMP T35.
        .
        .
        .
        LOC L21.
(2)     LOAD INT TEMP T35.
        CMP(N) INT A INT 0.
        JMP NE R L18.
        RELEASE INT NONREC T35.
```

Figure 3. Another loop that does not work

The difficulty is 'standardization'. In the example of Figure 2, let us assume that some sequence of actions has caused the top of the stack to be in some register R2. Assuming also a solution to the state of the stack conundrum, it is likely that at LOC L18 this register will still be the top of the stack. The problem is to guarantee that at LOC L21 the registers assigned to holding the elements of the stack are precisely the same. Since the code in the body of the loop is quite arbitrary (perhaps containing a procedure call requiring the registers to be flushed and restored) the standardization can only be done by rearranging the stack to a predefined order at each branch (which would in general be costly unnecessary execution overhead) or remembering the arrangement at each branch (a vast translation task) so that the standardization can be performed where necessary (still costly in some situations).

The same difficulty afflicts the use of named temporaries. If a named temporary variable is held in a register, it is always possible to generate a code sequence containing sufficient other named or anonymous temporaries to cause the register to be flushed before the final use of the temporary, and so lead to the possibility of having it reassigned to a different register. In Figure 3, were this to happen, the reference to the named temporary at position (1) would not use the same register as position (2), although they are only three Janus instructions apart coming around the loop.

Our present solution is to have each LOC pseudo void the stack, and not to permit a named temporary to be reserved past a LOC. This is undoubtedly overkill and prohibits various elementary types of optimization in the translation to target code. However, the alternative to this inefficiency is the expense of performing global flow analysis in the Janus translators for each target machine. It is not only that this task is beyond the portable tools that we have, but that the implementation effort for each target would negate any advantages available from Janus-based portable systems.

The question of standardization of the accumulator (but not of the stack) was addressed in the Coleman machine. With only one or two registers involved the amount of information to standardize the position can be firmly bounded, and the amount of target code to accomplish it need not be excessively expensive in execution time.

The above considerations also apply to the condition code register. The function of this register is unchanged from the Coleman machine, but now we specify that the value is preserved on the unsuccessful ('fall-through') branch of a test, and is not available at all on the successful branch (where the Coleman machine allowed it to be preserved on either or both paths at will).

UNRESOLVED QUESTIONS

Many aspects of the design of a universal intermediate language are yet to be researched. There are a number of questions of which we are already aware; the following is a sample.

Partword addresses

Janus at this time has only one address mode, and it is our tacit assumption that values of this mode provide sufficient information to access all objects of interest. However, we realize that if this set of objects were to include characters occurring within strings, or other objects packed without regard to word alignment, then large overheads could be incurred. Our feeling is that a second address mode would suffice.

Comparison of records

We have a comparison operator, CMPM, that performs a relation test on two specified memory areas (two records, two strings, etc). This operator refuses to take a midposition between high-level abstraction and simplicity of implementation. Ideally, its implementation is a loop of word or byte comparisons, but if the memory areas include items of differing modes, correct operation may require a mixture of different comparisons. This would entail either expansion of the operator into a sequence of comparisons or the generation of a template with interpreted execution, both approaches involving complex translation and inefficiencies in execution. A further problem is that alignment of fields within the record may leave holes which could contain arbitrary information. These holes must be masked or skipped during comparison, or the entire area allocated to the record must be cleared before each use.

Exception conditions

We have introduced the TRAP instruction as a means of programming responses to abnormal situations (e.g. violation of an index bound) that arise in the course of a computation. TRAP has the semantics of a conditional subroutine call, invoking the support system if the condition code has a specified value. We wished to avoid an out-of-line jump for two reasons: one is that it is frequently desirable for the handling routine to have the address of the infraction for reporting purposes, and the other is that it is sometimes desired to resume the computation after noting or reporting the abnormality. In this latter case, if a jump to the handling routine were used, a jump back would be needed, necessitating a LOC label at the re-entry point. This LOC would void the operand stack, destroying the computation in progress and the usefulness of returning.

We recognize that many languages have similar exception conditions, but we are not sure that it is worth standardizing names for these conditions, and doubt that uniform handling routines can be defined.

Case statements

Many case statements as they appear in programs in high-level languages are better implemented as a chain of conditional statements. The compiler is in a good position to decide this in many circumstances, but if it does so, it hides the existence of the case construct from the Janus-to-target translator, which may in some other circumstances be in a better position to make the decision (e.g. existence of a particular hardware feature for multiway branching). We have not yet been able to formulate a Janus representation of the case construct that preserves sufficient information in a suitable form for the decision to always be left to the translator.

Loops

In specifying the Coleman machine, it was stated[1] that it would be desirable to have a loop construct, as many machines do have special facilities of which advantage may be taken. We know no better now how to do this than was known then, yet our experience provides two motivations: we would like to be able to implement certain loops with a counter in a register, but cannot because of the general difficulties with temporaries; we also realize that decomposing a loop into tests and umps is telling the translator 'how' rather than 'what'.

Feedback

Most conventional compilers rely upon some information about the target machine or even about actual code generated to modify the behaviour of some parts of their analyser modules. Such information is termed 'feedback'. In the interests of achieving machine-independence, we were willing to forego this luxury[2] but it is now clear that some feedback is necessary (c.f. the case of the real denotations). We are now attempting to find a suitable set of parameters, 'environmental enquiries' if you will, for guiding a compiler in its task, that can be established *a priori*, still leaving it free from needing to know details of the actual target code generated.

SOME RESULTS

We mentioned above that we had been involved in the implementation of a set of routines to provide basic mathematical functions. In this section, we shall present some results of that implementation as an indication of the quality of portable software using the Janus concept. There are several reasons for our choice of this particular project as a benchmark:

—basic function routines are implemented on virtually all computers, and hence provide a broad basis for comparisons;

—the routines for a given machine are normally hand-coded, and thus present an objective measure of resource requirements uncontaminated by compiler ineptitude;

—numerical accuracy can be used as a standard measure of the quality of the routine;

—the structure of the routines is such that basic block optimization is likely to be more important than global flow analysis in reducing time and space.

One might regard the last point as a liability, biasing the results in favour of Janus. This may be a correct assessment—certainly some obvious global optimizations are difficult in Janus, as discussed above. We feel, however, that the test does tell us a good deal about the viability of Janus. As Wulf and his co-workers have pointed out:[22] 'In the final analysis the quality of the local code has a greater impact on both the size and speed of the final program than any other optimization.'

As noted in the *Introduction*, the basic function routines were hand-coded in Janus and parameterized with respect to certain target machine characteristics. They are distributed to prospective implementors as a master deck consisting of specialization macros followed by code which calls these macros. The implementor's first step is a STAGE2 run using the master deck and one or two parameter cards describing the radix and precision of the target computer's arithmetic unit. This run creates a set of Janus modules tailored to the given arithmetic unit. (The parameters govern the overall choice of algorithm and the constants of the approximation; see Reference 13 for details.) The tailored Janus modules are then implemented on the target machine as any other Janus code would be implemented.

We also used STAGE2 as the implementation tool for these tests, and in particular employed the lower-level abstract machines J1 and J2[23, 24] as intermediate steps in the translation. This implementation technique is the simplest one currently available, and would probably be used for the initial evaluation of Janus on most computers: Machine-independent STAGE2 macros for translation of Janus to J1 and J1 to J2 are provided as part of the Janus distribution. The distribution also contains macros which have been developed for translation of J2 to several assembly languages. These macros can be used as skeletons to reduce the effort involved in creating macros for a new machine to the order of 1–2 man-weeks.

At the present time we have implementations of SQRT, SIN/COS, ALOG/ALOG10 and EXP on three computers: a CDC6400, a General Precision L3055 and a Xerox Sigma 3. The major characteristics of these machines are summarized in Table I. The machines are dissimilar, and the algorithmic techniques of this set of routines are sufficiently varied that it is expected that the observed trends will be repeated as routines are added to the package and the number of implementations increased.

Table I. Machines used in testing basic functions

	Control Data 6400	General Precision L3055	Xerox Sigma 3
Registers	8 general purpose	1 accumulator	1 accumulator
Radix	2	10	2
Precision (radix digits)	48	8	31
Machine epsilon*	0.71E–14	1.0E–7	0.93E–9
Floating-point	Hardware	Hardware	Software
Word	60 bits	8 characters	16 bits

* Machine epsilon, *eps*, is the smallest positive floating-point number such that $1.0E0+eps\neq 1.0E0$.

Table II shows some comparisons of the machine-independent implementations relative to the standard FORTRAN libraries for each of the tested machines. The overall comparisons displayed are very largely representative of the figures for the individual routines. (More detailed measurements than these have been made.[25])

Table II. Preliminary comparisons for the mathematical routine package (containing SQRT, SIN/COS, ALOG/ALOG10 and EXP)

	CDC6400	L3055	Sigma 3
Standard library*	2.9E–14 (ALOG10)	166600E–7 (SIN)	581400E–9 (LOG)
Machine-independent library*	3.4E–14 (ALOG10)	39E–7 (EXP)	60E–9 (EXP)
Total size (relative to the library) (%)	105	107	163
Average execution time (%)	141	114	172

* Worst case maximum relative error.

The accuracy of the machine-independent routines is uniformly good. In fact, our testing procedure has turned up some surprisingly large errors in the standard libraries. Examination of the code of the library routines indicates that these errors can be fixed quite simply, but this apparent simplicity is a result of the expertise acquired with the machine-independent algorithms. The major benefit of the portable package is the ability to distribute this expertise in the form of Janus code.

Careful examination of the library routines for the Control Data 6400 indicates that their algorithms are quite different from those in the machine-independent package. The proportion of floating-point operations is lower, with much of the work carried out using bit manipulation and integer arithmetic. Such specialization is impossible given the goals of our package, and thus the execution-time penalty constitutes a legitimate cost of portability.

The exceedingly unfavourable comparisons on the Sigma 3 are due to two identified causes: (1) no advantage has been taken (as yet) of the 'short-form' floating-point software operations, thus making each such operation three words longer than those used in the standard library. This optimization is straightforward and will reduce the total size of the package to about 117 per cent of the size of the corresponding library routines; (2) the time figure is heavily influenced by the SQRT routine (1·8 ms for the Janus version compared with 0.26 ms for the standard library). This differential is due to the use of fixed-point arithmetic in the standard SQRT rather than floating-point. We are currently working on fixed-point implementations for all routines.

Starting from the Janus text, each implementation was obtained in less than 2 man-weeks of effort. The major part of this effort is defining the J2 machine, and is a task independent of the package size. Although the manpower investment in the algorithmic design and Janus coding is considerable (and proportional to the package size) the low incremental cost per implementation would allow a favourable amortization over a reasonable number of machines. There is no doubt that the machine-independent mathematical package would be an adequate library for most installations, and (should the penalties be critical for a given application) could certainly serve while hand-coded routines were developed.

CONCLUSION

The design of an abstract machine is in part a search for suitable models of computation. The components of the Janus abstraction that have given us the least difficulty are those developed from theoretically sound models. The operand stack (reverse Polish expression evaluation), the memory (VDL state) and recursive procedures (contour model) have all been found to be free of awkward exceptions, and map in regular ways into other architectures. Where we have no overall theoretical base upon which to build (e.g. global linkage, non-recursive procedure activation, looping and branching) we have had to work our way through a number of trials to arrive at our present definitions.

In devising a model, the 'how versus what' principle must be carefully considered. Since realizing the importance of this principle, we have made consistent appeal to it. Although it does not help to derive better models, it does give us a criterion for rejecting the less useful alternatives. Our experience also indicates that any proposed model must be extensively tested with a variety of language/machine combinations; it is all too easy to overlook critical details when building models on paper.

We are satisfied that the Janus approach to machine-independence is viable, and can be used as a tool in porting software. The most critical area at the moment is the lack of support routines for implementation. Our current research is concerned with providing such support: basic function routines, I/O and formatting routines, memory management. We are also attempting to develop a standard operating system interface for Janus, and investigating alternatives to STAGE2 as code generation tools.

Above all, we are convinced that the definition of the Janus machine serves as a powerful conceptual tool in identifying problems in machine-independence and portability.

ACKNOWLEDGEMENTS

Many people have provided feedback on the design of Janus. The detailed comments of Hendrik Boom (Mathematisch Centrum), Bernd Eichenauer (GPP, Munich), R. Haentjens (Ecole Militaire, Brussels) and Christopher Holt (Argonne National Laboratory) were particularly valuable. The Sigma 3 implementation used to obtain the results reported in this paper was carried out by T. L. Spear of the Computer Laboratory for Instruction in Psychological Research, University of Colorado.

This work was supported, in part, by the National Science Foundation under Grant No. NSF MCS74–20427 A01.

REFERENCES

1. S. S. Coleman, P. C. Poole and W. M. Waite, 'The mobile programming system: Janus', *Software—Practice and Experience*, **4**, 5–23 (1974).
2. S. S. Coleman, 'Janus: a universal intermediate language', *PhD. Thesis*, University of Colorado (1974). *NTIS PB 232 923/AS.*
3. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract machine modelling to produce portable software—a review and evaluation', *Software—Practice and Experience*, **2**, 107–136 (1972).
4. T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall Inc., Englewood Cliffs, N. J., 1975.
5. W. M. Waite and B. K. Haddon, 'A preliminary definition of Janus', *SEG-75-1*, Software Engineering Group, Department of Electrical Engineering, University of Colorado (revised September 1976).
6. W. M. Waite, 'The mobile programming system: STAGE2', *Comm. ACM*, **13**, 415–421 (1970).
7. W. M. Waite, *Implementing Software for Non-numeric Applications*, Prentice-Hall Inc., Englewood Cliffs, N. J., 1973.
8. N. Wirth, 'The programming language Pascal', *Acta Informatica*, **1**, 35–63 (1971).
9. K. R. Elz and P. C. Poole, 'An implementation of Janus', *Proc. Seminar/Workshop on Programming Language Systems*, Australian National University, Canberra, (1977).
10. H. Boom, 'Some preliminary experience with Janus', *unpublished report*, Mathematisch Centrum, Amsterdam, The Netherlands (1976).
11. A. van Wingaarden *et al.*, 'Revised report on the algorithmic language Algol 68', *Acta Informatica*, **5**, 1–236 (1975).

12. W. J. Cody, Jr., 'Software for the elementary functions', in *Mathematical Software*, Academic Press, New York, 1971, pp. 171–186.
13. W. J. Cody, Jr. and W. M. Waite, 'A software manual working note #1, preliminary draft of chapters 1–4d' *Technical Memorandum No. 321*, Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois (1977).
14. P. Lucas *et al.*, 'Method and notation for the formal definition of programming languages', *TR 25.087*. IBM Laboratory, Vienna (1968).
15. P. Wegner, 'The Vienna definition language', *ACM Comput. Surv.* **4**, 5–63 (1972).
16. E. W. Dijkstra, 'Recursive programming', *Numerische Mathematik*, **2**, 312–318 (1960).
17. J. B. Johnston, 'The contour model of block structured processes', *Proc. Conf. Data Structures in Programming Languages*, *SIGPLAN Notices*, **6**, 55–82 (1971).
18. English Electric–Leo–Marconi, *KDF9 Programming Manual*, Kidsgrove, Stoke-on-Trent, Staffordshire, 1964.
19. W. M. Waite, 'Janus', in *Software Portability* (Ed. P. J. Brown), Cambridge University Press, Cambridge, England, 1977, 277–290.
20. N. Wirth, 'Design and implementation of Modula', *Software—Practice and Experience*, **7**, 67–84 (1977).
21. D. W. Matula, 'In-and-out conversions', *Comm. ACM*, **11**, 47–50 (1968).
22. W. A. Wulf *et al.*, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.
23. W. M. Waite, 'Janus memory mapping: the J1 abstraction', *SEG–76–1*, Software Engineering Group, Department of Electrical Engineering, University of Colorado (1976).
24. W. M. Waite, 'Janus stack mapping: the J2 abstraction', *SEG–78–1*, Software Engineering Group, Department of Electrical Engineering, University of Colorado (1978).
25. W. M. Waite and B. K. Haddon, 'The performance of machine-independent basic function routines' (in preparation).