ON "PASCAL", CODE GENERATION, AND THE CDC 6000 COMPUTER

BY

NIKLAUS WIRTH

STAN-CS-72-257
FEBRUARY 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY
Abstract:

"Forth AL" is a general purpose programming language with characteristics similar to ALGOL 60, but with an enriched set of program- and data structuring facilities. It has been implemented on the CDC 6000 computer. This paper discusses selected topics of code generation, in particular the selection of instruction sequences to represent, simple operations on arithmetic, Boolean, and powerset operands. Methods to implement recursive procedures are briefly described, and it is hinted that the more sophisticated solutions are not necessarily also the best. The CDC 6000 architecture appears as a frequent source of pitfalls and nuisances, and its main trouble spots are scrutinized and discussed.
On "PASCAL", Code Generation, and the CDC 6000 Computer

1. Introduction

This set of notes has a dual purpose. It is on the one hand directed to the user of the PASCAL compiler system who would like to gain some insight into the machine code which is generated for various basic operations. It is even recommended that he study these notes carefully, because their understanding may prevent him from certain pitfalls which are inherent in the use of the CDC 6000 computer [1].

On the other hand the notes may be of interest to compiler writers in general, because they point out some problems and dilemmas and our choices of solutions. It becomes apparent that the choice of the code to be generated is crucial for a good compiler system, and that it is far from trivial as is usually believed.

The true purpose of a higher-level language is that it allows a programmer to conceive his algorithms in terms of some convenient abstractions. For instance, he is given the opportunity to think in terms of familiar notions of numbers, of relations, and of repetitions, instead of having to express his program in terms of bitstrings, arithmetic instructions, and transfers of control. However, these abstractions are only truly useful, if he can assume that his implementation observes all the properties which are commonly attributed to these abstractions, or else if it automatically issues a warning. As an example, when dealing with numbers in a high-level language, one should like to assume all the common axioms of arithmetic to hold. Of course this is not possible, since computers can only represent finite ranges of values. So one expects to receive a warning, if an operation
has trespassed the limits imposed by the implementation and an operation generates a result not in accord with the rules governing the abstraction. So the system is expected to provide an error indication, e.g. if an overflow occurs in an addition, if a value is being assigned which lies outside the specified range of values of variables, or if an array index is used which lies outside the defined limits.

Unfortunately, such potential warnings require the execution of additional instructions, which in general is costly. As far as range checking is concerned, they can be requested to be generated by the compiler for run-time execution by enabling so-called options. (The A-option generates assignment range checks, the X-option index checks.) They are relatively costly, but may speed up the finding of logical mistakes a great deal.

As far as irregularities of the arithmetic are concerned, one has become used to receive these warning signals automatically from the hardware, particularly because they are easily generated by the hardware, whereas a solution to detect overflow by software is usually beyond any reasonably economical feasibility. Unfortunately, the CDC computer fails to satisfy even the most modest expectations in this respect, and the effort to provide a system with security in the above sense was therefore a series of constant frustrations. Equally disappointing are some of the "features" of its floating-point arithmetic instructions.

One can go only a relatively short distance in trying to correct mistakes of the hardware by means of software; otherwise a system becomes ridiculously inefficient and will not be used by conscientious programmers who are willing to take the peculiarities of a hardware into account and guarantee safety of their algorithms by analytical
rather than experimental means. And this would have been against the intentions of PASCAL. So all that can reasonably be done is to elucidate the shortcomings and limitations of the hardware that are still transparent through the "software cover", and to make the programmer fully aware of them. And this is the purpose of this note.

It concerns itself with the simple operations of integer and real arithmetic, with Boolean operations and with powersets. The reader is supposed to be familiar with the CDC COMPASS notation. The operands are usually assumed to have been brought into the X1 and X2 registers. (If they were loaded into other registers, a corresponding renumbering is necessary which is, however, irrelevant to the operation itself). Registers X1-X5 are used as a stack for intermediate results, whereas X0 is used exclusively as local work register.

Section 6 deals with the topic of implementing recursive procedures and the addressing of local variables. Although the general techniques are well-known, analysis of possible solutions and their experimental comparison yielded some noteworthy results. It is shown that attempts to make full use of available hardware features such as base registers may not necessarily lead to an optimal performance. Again, the instruction set of the CDC computer is hardly optimal to implement mechanisms for recursive procedures. Conspicuously absent is a subroutine jump instruction which leaves the code invariant (reentrant).
2. **Integer Arithmetic**

Data of type integer or of subranges thereof are represented by fixed-point binary numbers. Addition and subtraction are represented by the

\[ \text{IXi} \quad X_j + X_k \]

instructions. Other operations are implemented by short sequences of instructions, as outlined below.

2.1 **Multiplication**

Due to a recent change of the hardware, fixed-point multiplication can be performed by a single

\[ \text{DX1} \quad x_1 * x_2 \]

instruction. It should, however, be noted that this instruction is essentially a floating-point instruction, and yields incorrect answers for fixed-point operands with \(|x| \geq 2^{48}\). This can be regarded as an overflow condition which is, alas, neither trapped nor indicated by the computer. A "safe code", checking against all imposed limits of operands and result, is quite elaborate and uneconomical by any standards, and was therefore not implemented.

If one of the operands is a constant \(C\) being representable as either

1. \(c = 2^n\) \(\{ (2, 4, 8, 16 \ldots) \)  
2. \(c = 2^m + 2^n\) \(m > n\) \(\{ (3, 5, 6, 9, 10, 12 \ldots) \)  
3. \(c = 2^m - 2^n\) \(\{ (7, 14, 15 \ldots) \)
then the compiler generates the following code for the multiplication
of \( X_1 \) by \( c \):

1: \text{LX}_1 \ n \quad \text{multiply by } 2^n

2,3: \text{LX}_1 \ n
\text{BX}_0 \ X_1
\text{LX}_1 \ m-n
\text{IX}_1 \ X_1 + X_0

Again, overflow conditions are simply ignored. Case 3 yields only
correct results, if \( |X_1 \cdot 2^m| < 2^{59} \).

2.2 Division (div)

Integer division is represented by the instruction sequence

\[
\begin{align*}
\text{FX}_1 & \quad X_1 \\
\text{FX}_2 & \quad x_2 \\
\text{NX}_2 & \quad x_2 \\
\text{FX}_1 & \quad X_1 / X_2 \quad \text{divide} \\
\text{UX}_1 & \quad B7, X_1 \\
\text{LX}_1 & \quad B7, X_1 \\
\text{BX}_0 & \quad X_0 - X_0 \\
1x1 & \quad X_1 + X_0 \\
\end{align*}
\]

\} \quad \text{suppress neg. zero}

and suffers from the same basic shortcoming as multiplication: an
operand \( |x| \geq 2^{48} \) yields an incorrect result.

If the divisor is a constant \( c = 2^n \), the compiler again produces
an "optimized" code, performing division by shifting. Unfortunately,
a single right shift instruction is unsatisfactory, because it may
generate a "negative" zero as result. Negative zeroes, however, must
not be allowed to occur, since comparisons may yield wrong answers if
applied to them. Thus, the optimized division is implemented as

\[
\begin{align*}
\text{AX}_1 & \quad n \quad \text{divide by } 2^n \\
\text{BX}_0 & \quad X_0 - X_0 \quad \text{suppress} \\
\text{IX}_1 & \quad X_1 + X_0 \quad \text{negative zero} \\
\end{align*}
\]

Note that the unconditional generation and addition of a zero can be
accomplished with a code that is not only shorter than a conditional jump, such as

```
AX1 n
NZ X1,L
SX1 BO
L ...
```

but also avoids the insertion of padding instructions (NOPs) for word boundary alignment.

The D-option provides an additional security measure against division by zero. It causes the compiler to insert a

```
ZR X2, error
```

jump instruction preceding every division instruction. (This applies to the Modulus operation as well.) It is particularly recommended in the case of integer division, where the actual divide instruction generates a "floating-point infinity" value, which is incorrectly treated by the subsequent conversion instructions and thereby represents a senseless result.

2.3 Modulus (mod)

The modulus or remainder operation is defined as

```
x \mod y = x-(x \div y) \times y
```

As it involves integer multiplication and division operations, it suffers again from the same deficiencies of the 6000 arithmetic. Its corresponding code is:

```
PX0 X1
PX6 x2
NX6 X6
FX6 X0/X6
UX6 B7,X6
LX6 B7,X6
DX6 X6\times X2
LX1 X1-X6
```
2.4 **Sign inversion**

The use of one's complement representation for negative numbers makes again the most obvious choice of code

\[ \text{BX1} \quad -x_1 \]

unsatisfactory, because it might generate a "negative zero". So we use

\[ \text{BX0} \quad x_0 - x_0 \]
\[ \text{IX1} \quad x_0 - x_1 \]

2.5 **Comparisons**

Since the computer does not offer a compare instruction, subtraction has to be used; this has primarily the disadvantage of generating wrong results in the case of overflow. The cases of testing for equality and inequality are handled correctly, because the one's complement addition generates an end-around-carry in the case of "negative overflow", thus maintaining a result indicating inequality. Note that the Boolean subtraction

\[ \text{BX1} \quad x_1 - x_2 \]

cannot be used, because a comparison of \( x_1 \) and \( x_2 = -x_1 \) would yield a zero result, thus indicating equality.

Whereas equality testing is "safe" with the

\[ \text{IX1} \quad x_1 - x_2 \]

instruction ignoring overflows, this is not the case for the tests of ordering \((x_1 < x_2)\) by subtraction and subsequent inspection of the sign bit. The reason is that if overflow occurs, i.e., \(|x_1 - x_2| > 2^{59}\), then the sign bit will be the opposite of the true sign. This situation is quite hopeless, since overflow is in no simple way detectable on this machine. In order to obtain a (sign) bit representing the relation
x < y for any values x, y, the following algorithm can be used:

1. Compare the signs of the two operands.
2. If they are different, then the result is obvious.
3. If they are equal, the subtraction x - y can be performed without danger of overflow, and x - y < 0 is the result.

A minimal instruction sequence to perform these operations and avoiding the use of undesirable jump instructions is

\[
\begin{align*}
&\text{BX0 } X1-X2 \quad \text{compare sign bits} \\
&\text{IX2 } X1-X2 \\
&\text{BX1 } X0\times X1 \quad \text{if unequal, choose sign of } X1 \\
&\text{BX2 } -X0\times X2 \quad \text{if equal, choose sign of } X1-X2 \\
&\text{BX1 } x1+x2
\end{align*}
\]

Now the sign bit of X1 is 1, if X1 < X2, and 0 otherwise. Still, the effort to perform a faultless comparison is formidably cumbersome, and the PASCAL compiler does not generate it. The programmer is left with the responsibility to verify that for every comparison of x and y, \(|x-y| < 2^{59}\).

2.6 Taking the absolute value (ABS)

The code used to take an absolute value is designed to avoid jump instructions, not only because they are long and slow, but because they usually introduce NOP instructions for alignment.

\[
\begin{align*}
&\text{Bxo } X1 \\
&\text{AX0 } 59 \quad \text{generate 60 sign bits} \\
&\text{BX1 } X0-X1
\end{align*}
\]

2.7 Testing for even or odd (ODD)

Since one's complement representation is used for negative numbers, the least significant bit of the operand must be compared with its sign bit:
This leaves the sign-bit of \( x_1 \) equal to 1, if \( x_1 \) was odd, and 0 otherwise.

The compiler "optimizes" in the case of \( \text{ODD}(x) \) with \( x \) being of a \text{subrange} type with only non-negative values. It then generates the single instruction

\[ \text{LX1 59} \]

### 2.8 Summary

The foregoing explanations reveal that the absence of any overflow indication makes analytical verifications necessary that guarantee the non-occurrence of these conditions. An effective aid in experimental testing is the \text{A-option}, causing interval check instructions to be generated with every assignment to a variable that is declared to be of a \text{subrange}'type. The \text{A-option} is activated by the "comment"

\[ \{$A^+ \ldots \}$ \]

and causes the code for an assignment to a variable

\[ \text{VAR V: a..b} \]

to become:

\[
\begin{align*}
\text{SX7} & \quad * & \text{location identification for error trap} \\
\text{SX1} & \quad a \\
\text{LX0} & \quad x_6-x_1 \\
\text{SX1} & \quad b \\
\text{BX1} & \quad x_1-x_6 \\
\text{BX0} & \quad x_1+x_0 \\
\text{NG} & \quad x_0,\text{error} & \text{jump to error routine} \\
\text{SA6} & \quad V
\end{align*}
\]

It should be noticed that unfortunately the attractive and shorter code sequence
cannot be used, because the instructions

\[ SX_i \quad X_j + K \]

perform an 18-bit arithmetic ignoring the leading 42 bits of the register \( X_j \) which -- of course -- is not in the spirit of a check.

This ignoring rather than checking of the leading bits in 18-bit arithmetic is the reason why the so-called "increment" instructions cannot be used by the PASCAL compiler, except in the following special circumstance: if a variable \( x \) is declared of a subrange whose limits are both less than \( 2^{17} \) in absolute value, then the assignment statement

\[ x := x + k \]

is compiled as

\[ SA1 \quad x \]
\[ SX6 \quad X1 + k \]
\[ SA6 \quad x \]
3. Floating-point Arithmetic

The PASCAL compiler uses the complete set of F-instructions for arithmetic with values of type "real". Comparison is performed by subtraction due to the lack of a compare instruction. This is possible without handicap since the occurrence of overflow generates a signed "infinity" value, but no immediate trap. Sign inversion is represented by

\[
\begin{align*}
\text{BX0} & \quad X0-X0 \quad \text{generate zero} \\
\text{IX} & \quad X0-X1 \\
\end{align*}
\]

and the absolute value function by

\[
\begin{align*}
\text{BX0} & \quad X1 \\
\text{AXO} & \quad 59 \\
\text{BX1} & \quad X0-X1 \\
\end{align*}
\]

Arithmetic with the F-instruction possesses some peculiar properties which will briefly be reviewed, and has for instance the consequence that \(x-y = 0\) does not necessarily imply \(x = y\), if the difference is computed by an F-instruction. The trouble arises from the fact that F-arithmetic truncates without rounding, and F-addition truncates without post-normalization. Every addition is therefore compiled into two instructions:

\[
\begin{align*}
\text{FX} & \quad X1+X2 \quad \text{add/subtract} \\
\text{NX} & \quad X1 \quad \text{post-normalize} \\
\end{align*}
\]

If the two values

\[
\begin{align*}
a &= 1720 \ 40\ldots00B & = 1.0 \\
b &= 1717 \ 17\ldots77B & = 1.0-2^{-48}
\end{align*}
\]

are compared by subtraction

\[
\begin{align*}
\text{FX} & \quad X1-X2 \quad a-b \\
\end{align*}
\]

the result is
where the slash marks the separation between the lower and the upper half of the 96-bit accumulator. The result is 0 although the two operands were different.

Notice that subtracting 0.5 from both a and b, and then computing their difference, yields

\[
\begin{align*}
a - 0.5 &= 0.5 : 1717 \ 40... \ 00 \\
b - 0.5 &= 0.5-2^{\text{-48}} : 1716 \ 77... \ 76
\end{align*}
\]

\[
\begin{align*}
1717 \ 40... \ 00 / 00... \ 0 \\
-1717 \ 37... \ 00 / 00... \ 0 \\
1717 \ 00... \ 01 / 00... \ 0
\end{align*}
\]

i.e., a difference which is not zero. Thus the result does not only depend on the true result, but also on the values of the operands. This unpleasant, property of the CDC F-arithmetic stems from the fact that automatic post-normalization is absent.

3.1 Rounding

It was at one time hoped that this defect could be avoided by letting the PASCAL compiler automatically generate R-instructions, which include a certain kind of rounding. However, R-arithmetic turned out to feature some even stranger properties, so that it was decided not to use R-instructions. In order to point these features cut, a brief review over R-arithmetic is necessary:
The R-instructions differ from the F-instructions only insofar as a 1-bit is appended to normalized operands before the arithmetic operation is performed. Thus for instance the subtraction of $b = 1.0 - 2^{-48}$ from $a = 1.0$ yields

\[
\begin{align*}
1720 & \quad 40 \ldots \quad 00 \quad / \quad 40 \quad \ldots \quad 0 \\
-1720 & \quad 37 \ldots \quad 77 \quad / \quad 60 \quad \ldots \quad 0 \\
& \quad 1720 \quad 00 \ldots \quad 00 \quad / \quad 60 \quad \ldots \quad 0
\end{align*}
\]

1-bits appended

which of course is still zero.

The principal defect with "CDC-rounding", however, is that its effect is unpredictably either the addition of $1/2$ or $1/4$ in the last position, because rounding takes place before instead of after normalization (which must again be performed by a separate instruction). The following example illustrates this, which is shown on hand of a five-bit number representation:

\[
\begin{align*}
16 &= 10000 / 1 \quad \text{inserted round-bits} \\
+17 &= 10001 / 1 \\
33 &= 100010 / 0 \\ & \quad 10001 / 0 = 34 \\
31 &= 11111 / 1 \quad \text{inserted round-bit} \\
+2 &= 00010 / 0 \\
33 &= 100001 / 1 \\ & \quad 10000 / 1 = 32
\end{align*}
\]

In the first case, the pre-rounding results in correct rounding of the not exactly representable $33$ to $34$, whereas in the second case pre-rounding has no effect.

The same phenomenon can be observed in the cases of multiplication and division. The following example again uses a five-bit number representation:
In the first case, the rounding effect is nil, leaving the inexactly representable value 180 be an unrounded 176; in the latter case the rounding effect transforms 180 into the value 184. (Suitable adjustment of exponents is not shown here.)

A method introducing proper rounding instead of "CDC-rounding" relies on the use of the D-instruction set [2]. Whereas the F-instructions yield the high-order 48 bits of the 96-bit accumulator, the D-instructions yield the low-order 48 bits with a suitably adjusted exponent, thereby allowing access to a double precision result.

Notice that it is an ingeniously efficient method to compute a double precision result by

1. computing the DP-result and dispose of the low half (F-instruction), then
2. computing the same again and dispose of the high half (D-instruction).
This computer allows it to be done in no other way!

The PASCAL compiler will generate the following code for floating-point operations, depending on the choice of the R-option:

<table>
<thead>
<tr>
<th>R-option</th>
<th>OFF</th>
<th>ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y$</td>
<td>$X + Y$</td>
<td>$Fx + x2$</td>
</tr>
<tr>
<td></td>
<td>$Nx + X1$</td>
<td>$Fx + x2$</td>
</tr>
<tr>
<td>$x \times y$</td>
<td>$X \times y$</td>
<td>$Fx \times x2$</td>
</tr>
<tr>
<td></td>
<td>$Nx \times X1$</td>
<td>$Fx \times x2$</td>
</tr>
<tr>
<td>$x \div y$</td>
<td>$X \div y$</td>
<td>$Rx \div X0$</td>
</tr>
<tr>
<td></td>
<td>$Nx \div X1$</td>
<td>$Rx \div X0$</td>
</tr>
</tbody>
</table>

Examples of addition/subtraction:

1. \(10^{-48} \quad 1720 \ 40\ldots\ 00/00\ldots\ 0\)
\(1717 \ 77\ldots\ 77/00\ldots\ 0\)
\(1720 \ 37\ldots\ 77/40\ldots\ 0\)
\(= 1640 \ 40\ldots\ 00\)

after addition of high and low

2. Take \(a = 1.0\) and \(b = 2^{-48}\), then subtract \(a-b\):

F-subtraction yields

\[a = 1720 \ 40\ldots\ 00/00\ldots\ 0\]
\[b = 1720 \ 00\ldots\ 00/40\ldots\ 0\]
\[1720 \ 37\ldots\ 77/40\ldots\ 0\]

which, after normalization, is

\[1717 \ 77\ldots\ 76 = 1.0-2^{-47}\]

R-subtraction inserts a 1-bit after the slash in the first operand, and thus yields the result

\[1720 \ 40\ldots\ 00 = 1.0\]
The combined use of F and D instructions yields the true result, because the normalization instruction left shifts the high order result to

\[ 1717\ 77\ldots\ 76 \]

whereafter a "rounded" addition is used to add the correction

\[ +1717\ 009\ldots\ 01 \]

yielding

\[ 1717\ 77\ldots\ 77 = 1.0 - 2^{-48} \]

3.2 Conversion from fixed to floating-point (integer to real)

Wherever a real operand is permissible, PASCAL allows the specification of an operand of type integer as well. However, the compiler is then forced to generate the necessary representation conversion instructions, which are not only time-consuming, but potentially hazardous. It is therefore recommended to avoid "mixed-mode" arithmetic expressions wherever possible. The generated conversion instructions are

\[
\begin{align*}
\text{PX} & \quad \text{B}0,\text{X}1 & \quad \text{pack with zero exponent} \\
\text{NX} & \quad \text{B}0,\text{X}1 & \quad \text{normalize}
\end{align*}
\]

The result of this conversion is wrong, whenever the integer operand in X1 is larger or equal to \( 2^{48} \) in absolute value, since the exponent bits are simply ignored by the P instruction. A test to verify that the operand is within bounds could be compiled as

\[
\begin{align*}
\text{Bxo} & \quad \text{X}1 \\
\text{AX} & \quad 48 \\
\text{NZ} & \quad \text{X}0,\text{error}
\end{align*}
\]

but is easily seen to be more costly than the conversion itself.
3.3 Conversion from floating to fixed-point (real to integer)

PASCAL does not provide for any implicit real to integer conversion. However, the standard function \texttt{TRUNC}(x) allows to truncate the fractional part of a real number. The used code is:

```
UX1  B7,X1
LX1  B7,X1
BY0  X0-X0       \text{avoid negative zero}
IX1  X1+X0
```

The result of this conversion is again wrong, if \(|x| \geq 2^{138}\).
4. **Boolean Operations**

The standard type Boolean is defined in PASCAL as

```pascal
    type Boolean = (false, true)
```

Since the values of all scalar types are mapped onto the integers $0, 1, 2, \ldots$, the values `false` and `true` are represented by the numbers 0 and 1 respectively.

The operations $\land$ and $\lor$ are implemented by the Boolean AND and OR instruction, namely

- $\text{BX}1 \ x1\times x2$ and $\text{BX}1 \ x1+x2$

Negation is performed by

- $\text{MX}0 \ 59$
- $\text{BX}1 \ -x0-x1$

If a relation has to be assigned to a Boolean variable, e.g.

- $b := x < y$

then a sequence of instructions is necessary to obtain a 0 or 1 value. Again every effort is made to avoid the use of jumps. The following code is used in the above assignment; leaving a Boolean value in `X1`.

- $\text{FX}1 \ x1-x2$ $x-y$
- $\text{MXo} \ 1$
- $\text{BX}1 \ x0\times x1$ Extract sign bit
- $\text{LX}1 \ 1$ move it to correct position

Analogous code is generated for the relations $>$, $\leq$, and $\geq$. But unfortunately the equality relations cannot be reasonably implemented without a jump; in the assignment

- $b := x = y$

the following instructions are generated:
Boolean comparisons, although occurring rather infrequently, are treated as special cases, because a simpler and shorter code is applicable:

\[
\begin{align*}
\text{p} < \text{q} & \quad \text{BX1} \quad -x_1 \cdot x_2 \\
\text{p} \leq \text{q} & \quad \text{BX1} \quad -x_2 \cdot x_1 \\
\text{Mx0} & \quad 59 \\
\text{BX1} & \quad -x_0 \cdot x_1 \\
\text{p} \neq \text{q} & \quad \text{BX1} \quad x_1 \cdot x_2
\end{align*}
\]

The remaining three relations are compiled analogously.
5. **Powerset Operations**

PASCAL 6000 restricts `powerset` types to be built only on base sets with less than 59 components. This allows a `powerset` value $S$ to be represented by one "word", in which the $i$-th bit indicates the presence (1) or absence (0) of the element $i$ in $S$.

5.1 **Generation of the Singleton Set** \([i]\)

Assume that $i$ is loaded into register $X1$, then

\[
\begin{align*}
SB7 & \quad X1 \\
SX1 & \quad 1 \\
LX1 & \quad B7,X1
\end{align*}
\]

Notice that the numbering of bits starts with 0 at the low order end. This choice was made in order to be able to load `powerset` constants with small valued components (less than 18) by a single `SX1` instruction.

5.2 **Set Intersection, Union, and Difference**

These three operations are implemented by a single instruction

\[
\begin{align*}
\text{intersection} & \quad BX1 \quad x1 \cdot x2 \\
\text{union} & \quad BX1 \quad x1 + x2 \\
\text{difference} & \quad BX1 \quad -x2 \cdot x1
\end{align*}
\]

5.3 **Set Membership** \((\text{in})\)

The relation $i$ in $S$ is implemented by shifting the bit representing $i$ into the sign position which can be tested:

\[
\begin{align*}
SB7 & \quad X1 \quad 1 \\
AX1 & \quad B7,X2 \quad S \\
LX1 & \quad 59
\end{align*}
\]

If the expression $i$ is in the form of a constant $c$, then the compiler generates of course only the single instruction

\[
LX1 \quad 59-c
\]
5.4 Set Comparison

Sets can be compared for equality and inclusion. Equality is tested by a Boolean subtraction

\[ \text{BXO} \quad X_1 - X_2 \]

and a subsequent zero test. Note that the peculiar property of the zero test to recognize a word with either 60 zero-bits or 60 one-bits as a zero is responsible for the restriction that powersets may contain at most 59 instead of 60 elements. If sets with 60 components were allowed, then a full set and an empty set would not be distinguishable by a single subtraction followed by a zero-test.

Inclusion expressed as \( x \leq y \) and meaning \( x \subseteq y \), is implemented by the single instruction

\[ \text{BXO} \quad -X_1 \times X_2 \]

which is followed by a zero-test instruction. The same instruction is used for the relation \( x > y \), whereas strict inclusion \( (x \subset y) \) is not implemented.
Some Exercises Addressed to the CDC 6000 Expert.

1. Is the following code to represent the function \texttt{trunc(XL)} acceptable? If not, why?

\begin{itemize}
  \item \texttt{BX0} X0-X0
  \item \texttt{FX0} X0
  \item \texttt{FX1} XL+X0
  \item \texttt{UX1} B7,XL
  \item \texttt{NZ} B7, overflow
\end{itemize}

2. Is the following code for \texttt{XL mod X2} acceptable? If so, prove it.

\begin{itemize}
  \item \texttt{FX1} XL
  \item \texttt{FX2} x2
  \item \texttt{NX6} x2
  \item \texttt{FX6} XL/X6
  \item \texttt{Bx0} X0
  \item \texttt{FX6} X6+X0
  \item \texttt{DX6} X6*X2
  \item \texttt{FX6} XL-X6
  \item \texttt{UX1} X6
\end{itemize}

3. Why can the instructions

\begin{itemize}
  \item \texttt{BX0} XL-X2
  \item \texttt{ZR} X0, equal
\end{itemize}

not be used to represent a comparison \(XL = X2\)? Prove that

\begin{itemize}
  \item \texttt{1x0} XL-X2
  \item \texttt{ZR} X0, equal
\end{itemize}

always yields the correct action.
Implementation of Recursive Procedures

The language PASCAL has been carefully designed so that dynamic storage allocation is not required, with the following two exceptions:

1. Variables local to procedures may be allocated storage only when the procedure is called, and
2. Components of class variables are allocated storage by calling the standard procedure "alloc". An area of store is allocated to the entire class variable as soon as the procedure is called to which the class is local.

In this section we will briefly review the well-known techniques for handling recursive procedure calls and of allocating storage to their local quantities, and discuss the code selected to represent the procedure call mechanism.

Due to the first-in last-out nature of the hierarchy of activated procedures a stack may be used to allocate local variables. This is of great advantage, since storage retrieval is trivial in the case of stacks, resulting in low storage management overhead. We consider the set of local variables of each activated procedure as a record (often called "data segment") in the stack. Since their lengths may all be different, the most convenient method to thread the way back through such a stack is by constructing a chain of pointers linking the records. Every record then contains a "header" containing

1. the link to the previous record, and
2. the (frozen) program status (counter) of the calling procedure.

Variables are addressed relative to the origin of the record of which they are a part. The origin address is unknown at compile-time,
and must be determined at run-time. This can be done by descending through the link chain, until the desired record is reached. But how is the desired record recognized? The most straight-forward method which interprets the scope rules of an ALGOL block structure correctly is probably the following:

**Method I:**

1. Define the level of an object to be 1 greater than the level of the procedure to which it is local. The level of the main program is 0.
2. Indicate the level of each record (equal to the level of its components) in its heading.
3. Whenever an object on level \( i \) has to be accessed, the record containing it is found by descending down the chain of links until the first occurrence of a level indicator with value \( i \) is found.

This accessing method has the obvious drawback of inefficiency (and of not being applicable in the case of parametric procedures). A slight modification, however, improves efficiency and generalizes to parametric procedures.

**Method II:**

Instead of indicating levels explicitly in the record headings, a second link chain is constructed connecting each record \( A \) with its static ancestor, i.e., with the record \( B \) of the procedure in which \( A \) was declared locally. In order to distinguish the two link chains, the former is called the "dynamic link" and the latter the "static link".

An example of a state of computation is shown below for a given -- admittedly not very realistic -- program.
var v0;
procedure Q0 (procedure X);
  var w1;
  procedure Q1;
    var w2;
    begin w2 := w1+v0; X
    end;
  begin w1 := v0; Q1
  end;
procedure PO;
  var v1;
  procedure Pl;
    var v2;
    begin v2 := v1+v0; Q0(PO)
    end;
  begin v1 := v0; Pl
  end;
begin {main program} v0 := 0; PO
end.

Method III:

Although the use of a high-speed index register to represent the origin of the link chains improves access speed significantly, the process of descending down the static chain to the record (data segment) with the desired level is relatively time-consuming. An ingenious device to reduce access time was introduced by Dijkstra [4] and is now widely used in compilers for block-structured languages. The device
is an array of base addresses, called the Display $D$, which is at any time a copy of the static chain. If an object at level $i$ is to be accessed, the origin address of its data segment is quickly obtained as $D_i$. The method is particularly attractive for computers with a set of high-speed index registers which can be used as the Display. The price for this increase in access speed -- apart from the reservation of registers -- is the setting and updating of the Display each time a procedure is called and terminated. To be more specific, the necessary actions are as follows:

1. if an actual procedure of level $i$ is called, $D_i$ has to be set;
2. if control is returned from a procedure at level $i$ to one at level $j$, $(j \geq i)$, $D_i \ldots D_j$ have to be reassigned;
3. if a formal procedure at level $i$ is called from a procedure at level $j$, $D_i \ldots D_k$ have to be reassigned, where $k$ is the level on which the static link emerging from the calling and the called procedures merge. Since $k$ is not known at the time the procedure declaration is compiled, $k$ can be chosen as zero without significant loss in efficiency.

This scheme was used in the implementation of PASCAL 6000. It is described in Reference 4. Registers $B1 \ldots B5$ are used as the Display, $B5$ is the origin of the link chains, and $B6$ is the pointer to the top of the stack. The compiled instructions are the following:
Procedure call of P:

\[
\begin{array}{ll}
\text{SX7} & \text{L} \\
\text{EQ} & \text{P} \\
\hline
\text{L} & \text{SBj} \\
\text{SA1} & \text{B5} \\
\text{SB}(j-1) & \text{X1 I} \\
\ldots & \ldots \\
\text{SA1} & \text{X1} \\
\text{SB1} & \text{X1} \\
\end{array}
\]

save return address
and jump

update the display, if \( j \geq i \)

Procedure entry:

\[
\begin{array}{ll}
\text{PF} & \text{SBi} \text{X1} \\
\text{SA1} & \text{X1} \\
\text{SB(i-1)} & \text{X1} \\
\ldots & \ldots \\
\text{SA1} & \text{X1} \\
\text{SB1} & \text{X1} \\
\end{array}
\]

prolog, entry for calls
of formal procedures

update display

\[
\begin{array}{ll}
\text{PA} & \text{SA7} \text{B5+2} \\
\text{SX7} & \text{B(i-1)} \\
\text{SA7} & \text{B6} \\
\text{SX7} & \text{B5} \\
\text{SA7} & \text{B6+1} \\
\text{SB1} & \text{B6} \\
\text{SB5} & \text{B6} \\
\text{SB6} & \text{B6+L} \\
\end{array}
\]

save return address in header
save static link
save dynamic link
new display entry
T

top of stack, \( L = \) data segment length

Procedure exit:

\[
\begin{array}{ll}
\text{SB6} & \text{B5} \\
\text{SA1} & \text{B5+1} \\
\text{SB5} & \text{X1} \\
\text{SA1} & \text{B5+2} \\
\text{SB7} & \text{X1} \\
\text{JP} & \text{B7+0} \\
\end{array}
\]

reset top of stack
reset \( T \)
fetch return address and jump

Notice that global variables in the main program are assigned absolute addresses. Since \( B0 \equiv 0 \), they can be considered as based on \( B0 \).
In the first half of 1971, Prof. C. A. R. Hoare and his collaborators modified and bootstrapped the PASCAL compiler for the ICL 1966 computer [6]. One of the more significant alterations concerned the elimination of the Display, due to the fact that the ICL computer has no set of index registers that are available for a Display, and since the use of a Display was not considered to be an advantage, in this case. During a visit of Prof. Hoare in July 1971, he suggested that maybe even with a register set available for the Display, the benefits gained should be investigated. His suggestion was certainly valid, since variables either global or local to the most recently called procedure could be accessed with the same speed even without a Display. Thus the gain from a Display is limited to faster access of objects at intermediate levels, while the price is the updating at every call regardless of whether such objects are accessed or not. A superficial look at the PASCAL compiler itself showed that accesses to such intermediate level objects were indeed relatively rare, and it was decided to generate a version that would not use a Display (Method II). This version still uses the address register B5 as origin of the link chains (and base address of the most local data segment) and B6 as pointer to the top of the stack. The generated code is:

Procedure call P

* X6 := base of environment of P

SX7 L
EQ P
L ...

28
Procedure entry:

\[
\begin{align*}
&\text{SXO } B5 \quad \text{pack and store} \\
&\text{LXO } 18 \quad \text{dynamic link and} \\
&\text{BX7 } X7+X0 \quad \text{return address} \\
&\text{SB5 } B6 \\
&\text{SA7 } B5+1 \\
&\text{SB6 } B7+L \\
&\text{SA6 } B5 \quad \text{stack pointer} \\
&\text{* } \\
\end{align*}
\]

Procedure exit:

\[
\begin{align*}
&\text{SA1 } B5+1 \quad \text{fetch and unpack} \\
&\text{SB6 } B5 \\
&\text{SB7 } X1 \quad \text{dynamic link and} \\
&\text{LX1 } 42 \quad \text{return address} \\
&\text{SB5 } X1 \\
&\text{JP } B7+0
\end{align*}
\]

Fetching an object \(x\) at level \(j\) from code at level \(i\):

\[
\begin{align*}
1) \quad j = 0: & \quad \text{SA1 } B0+x \\
2) \quad j = i: & \quad \text{SA1 } B5+x \\
3) \quad 0 < j < i: & \quad \text{SA1 } B5 \\
& \quad \text{SA1 } X1 \quad \text{repeated } i-j-1 \text{ times} \\
& \quad \text{SA1 } X1+x
\end{align*}
\]

A comparison of the codes generated by the two compilers shows that gains and losses of execution speed should be measured, but also those of code length. The shorter codes for procedure entry (2 - 2\(\frac{1}{2}\) words vs. 4 - 6 words), procedure exit (2 vs. 3 words), and procedure calls (no updating of display) are very attractive, particularly in a compiler where space is more on a premium than time. (It should be noted that the instructions marked with an asterisk can be omitted in the call or the entry code of procedures declared on the first level). Of course it must be kept in mind that the decision about which compiler is to be preferred depends not only on the weighting of space vs. time, but even more on the programs to be processed. But it is obvious that if the
The majority of these programs rarely use nested procedure declarations, and often call procedures on the same level, then the compiler without Display is to be preferred. The compiler itself, although featuring nested procedure declarations, but seldom accessing intermediate level variables, belongs to this class. Comparisons of code generated by the two compiler versions produced the following results:

1. The efficiency of codes not using a Display is in the average slightly higher (the compiler itself runs about 1.5% faster).
2. The size of codes not using a Display is smaller (by about 4% measured on 25 sample programs, about 6% in the case of the compiler's code).
3. The compiler program itself is slightly less complex without Display.

This episode where a more sophisticated method was abandoned in favor of a simpler and more direct technique could well be added to the list of D. Knuth's examples of adverse influences of "computer science" on "computer usage" [5]. Their common characteristic is that improved methods are adopted without closer inspection of the nature and direction of the improvement, and without analysis of the circumstances to be improved. An interesting fact is that the Burroughs B5500 computer -- specifically designed for ALGOL implementation -- did contain exactly the two base registers required to efficiently address objects at levels 0 and 1. Unfortunately, addressing of intermediate level objects was impossible due to the software; this deficiency was justifiably criticized. The remedy adopted in the successor B6500 was, however, not a correction of the deficient software, but the inclusion of a full set of high-speed registers to serve as Display.
Summary of the Main Trouble Spots of the CDC 6000 Architecture

1. Use of one's complement arithmetic. In order to keep comparisons simple and efficient, the occurrence of negative zeroes must be prohibited. (Note that PL and NG test the sign bit only.) Various optimizations are more cumbersome and less effective, because negative zeroes must be suppressed by additional instructions. Some instructions are themselves unsafe against the generation of -0!

2. No overflow check on fixed-point arithmetic. This lack is very serious and may cause wrong results in totally unexpected situations. Overflow check by software is prohibitive.

3. No compare instructions. The use of subtraction may cause wrong results, unless expensive precautions are taken.

4. Use of 48-bit multiplier and divider for fixed-point 60-bit numbers without warning of possible "overflow" of operands.

5. Floating-point addition and subtraction without automatic post-normalization.

6. Floating-point arithmetic with rounding of operands instead of rounding postnormalized results.

7. No subroutine jump instruction depositing the program counter P in an operand register, and no return jump loading P from a general operand register. This defect requires the use of 3 instructions each to jump and deposit a return address, and to retrieve it and return, whereas many other computers need only a single instruction for these purposes.
Conclusions

When considering these complaints, the reader should bear in mind that this computer's architecture was conceived in the very early 1960's. The CDC 6600 machine was a very advanced design for a special purpose: fast number crunching. The design relied heavily on the use of several arithmetic units working simultaneously ("in parallel"). Integer arithmetic was considered as almost dispensible, and overflow interrupts as undesirable, because of the impossibility to mirror the present state of the entire machine by a simple program counter and of resuming computation. The use of simultaneously operating units is apparently also made responsible for the otherwise incomprehensible absence of post-normalization, namely because the unit for float-point addition does not contain a left-shift circuitry. A few years later, the CDC 6400 (and 6500) computers were announced; they were to have the same instruction set as the 6600, but only one conventional integrated arithmetic-logical unit. Although the "reasons" for the absence of interrupts and post-normalization had vanished, these "features" were retained in the name of compatibility. It was apparently considered most important that pitfall loaded programs could be transported to the new machines at no extra cost. This policy of staying "upward compatible with all previous mistakes" was sternly maintained when the successor to the 6000 series was announced in 1971.

This attitude, which is by no means atypical among computer manufacturers, makes it doubtful whether any progress toward more reliable and more efficient computing will ever be achievable. It does not seem so, until the computer consumers' attitudes will no longer justify the present manufacturers' policies. They, in turn, will not
change before they are made aware of the hidden cost involved in using the present equipment. I am convinced that the cost incurred by the programmers having to discover bugs the hard way by reprogramming repeatedly, and having to reexecute programs many times until they were believed to be correct, is incomparably higher than the reduction in cost due to staying compatible with outdated architectures. The project to develop the PASCAL compiler for the CDC 6000 computer unfortunately provided ample support for this conviction.

Acknowledgments

I am grateful to W. Kahan for pointing out some additional problems with the CDC floating-point arithmetic as well as the method for obtaining correct rounding.
References


BEGIN { REAL ARITHMETIC 3
x := 1.0; Y := x + 3.14159; Z := x*y + x/y;
x := x + (Y + (Z + (1.0 + x)));
a := ABS(+Y); Y := SQR(X); Z := -X;

$R+ ROUNDED REAL ARITHMETIC$
X := Y + Z; X := Y/Z;

INTEGER ARITHMETIC 3
I := 1; J := I + 100; K := I * J; K := I DIV J;
K := (-J) MOD K; J := SQR(J);
I := TRUNC(X); Z := I; X := I/J;

BOOLEAN ARITHMETIC 3
P := TRUE; Q := P AND Q;

P := x = Y; P := I = J; Q := P = Q;
P := x < Y; P := I < J; Q := P < Q;
P := x <= Y; P := I <= J; Q := P <= Q;
Q := MOD(I);

OPTIMIZATION OF INTEGER ARITHMETIC
I := I*8 + J*10;
J := I DIV 8 AN DIV 2; K := I MOD 16;
N := I + 100

END

005074
SA1 80+005004
SA2 80+005026

005076
BX6 X6
SA6 80+005004
NO

005077
SA1 80+005004
SA2 80+005026

005100
FX1 X1+X2
NX6 80,X1
SA6 80+005005

005101
SA1 80+005004
SA2 80+005005

005102
FX1 X1*X2
SA2 80+005004
NO

005103
SA3 80+005005
FX2 X2/X3
FX1 X1+X2

005104
NX6 80,X1
SA6 80+005006
NO

005105
SA1 80+005004
SA2 80+005005

005107
SA5 80+005004
FX4 X4*X5
NX4 80,X4

005110
FX3 X3+X4
NX5 80,X3
FX2 X2+X3
NX2 80,X2

005111
FX1 X1*X2
NX6 80,X1
SA6 80+005004
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>005145</td>
<td>SA1: B0+005004, UX1: B7,X1, 1x1: B7,X1</td>
</tr>
<tr>
<td>005146</td>
<td>OX0: X0=X0, IX6: X1=X0, SA6: B0+005001</td>
</tr>
<tr>
<td>005147</td>
<td>SA1: B0+005001, BX6: X1, PX6: B0,X6</td>
</tr>
<tr>
<td>005150</td>
<td>NX6: B0,X6, SA6: B0+005006, NO</td>
</tr>
<tr>
<td>005151</td>
<td>SA1: B0+005001, SA2: B0+005002</td>
</tr>
<tr>
<td>005152</td>
<td>PX2: B0,X2, NX2: B0,X2, PX1: B0,X1, NX1: B0,X1</td>
</tr>
<tr>
<td>005153</td>
<td>RX6: X1/X2, SA6: B0+005004, NO</td>
</tr>
<tr>
<td>005154</td>
<td>SX6: B0+000001, SA6: B0+005010</td>
</tr>
<tr>
<td>005155</td>
<td>SA1: B0+005010, SA2: B0+005011</td>
</tr>
<tr>
<td>005156</td>
<td>SA3: B0+005010, OX2: X2X3, MX0: 73</td>
</tr>
<tr>
<td>005157</td>
<td>OX2: X2=X0, BX6: X1=X2, SA6: B0+005011</td>
</tr>
<tr>
<td>005160</td>
<td>SA1: B0+005004, SA2: B0+005005</td>
</tr>
<tr>
<td>005161</td>
<td>IX0: X1=X2, MK6: 00, NZ: X0,005163</td>
</tr>
<tr>
<td>005162</td>
<td>SX6: B0+000001, NO</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>005163</td>
<td>SA6: B0+005010, SA1: B0+005001</td>
</tr>
<tr>
<td>005164</td>
<td>SA2: B0+005002, 1x0: X1=X2, MK6: 00</td>
</tr>
<tr>
<td>005165</td>
<td>NZ: X0,005166, SX6: B0+000001</td>
</tr>
<tr>
<td>005166</td>
<td>SA6: B0+005010, SA1: B0+005010</td>
</tr>
<tr>
<td>005167</td>
<td>SA2: B0+005011, BX1: X1=X2, MKO: 73</td>
</tr>
<tr>
<td>005170</td>
<td>BX6: X1=X0, SA6: B0+005011, NO</td>
</tr>
<tr>
<td>005171</td>
<td>SA1: B0+005004, SA2: B0+005005</td>
</tr>
<tr>
<td>005172</td>
<td>FX1: X1=X2, MX0: 01, %X6: X0X1, LX6: 01</td>
</tr>
<tr>
<td>005173</td>
<td>SA6: B0+005010, SA1: B0+005001</td>
</tr>
<tr>
<td>005174</td>
<td>SA2: B0+005002, 1x1: X1=X2, HXO: 01</td>
</tr>
<tr>
<td>005175</td>
<td>BX6: X0X1, LX6: 01, SA6: B0+005010</td>
</tr>
<tr>
<td>005176</td>
<td>SA1: B0+005010, SA2: B0+005011</td>
</tr>
<tr>
<td>005177</td>
<td>BX6: X1X2, SA6: B0+005011, NO</td>
</tr>
</tbody>
</table>
005200
SA1  80+005004
SA2  80+005005

005201
FX1  x2-x1
HKO  01
BX6  x1\times x0
LX6  01

005202
SA6  80+005010
SA1  004005001

005203
SA2  80+005002
1x1  x2-x1
MX0  01

005204
BX6  x1\times x0
LX6  01
SA6  80+005010

005205
SA1  80+005010
SA2  80+005011

005206
BX1  x2\times x1
MKO  73
BX6  x1-x0
NO

005207
SA6  80+005011
SA1  80+005001

005210
BX0  x1
LX0  73
BX1  x1-x0
MKO  01

005211
BX6  x0\times x1
LX6  01
SA6  80+005011

005212
SA1  80+005001
LX1  03

005213
SA2  80+005002
LX2  01
BX0  x2

005214
LX2  02
IX2  x2-x0
IX6  x1+x2
NO

005215
SA6  80+005001

005216
SA1  80+005001
AX1  03
BX0  x0-x0
IX1  x1+x0
NO

005217
SA2  804005007
AX2  01
IX6  x1-x2

005220
SA6  80+005002
SA1  80+005001

005221
BX0  x1
AX0  04
LX0  04
IX6  x1-x0

005222
SA6  80+005003
SA1  80+005001

005223
SX6  x14000144
SA6  80+005007

005224
SA1  85+x0
SB7  x1+x0
JP  87+x0

005225 17204000000000000000
005226 17216220771740156064