The MU5 compiler target language and autocode

P. C. Capon, D. Morris, J. S. Rohl, and I. R. Wilson

Department of Computer Science, University of Manchester, Manchester M13 9PL

In this paper the design of the software implementation language for the MU5 machine is considered. This has two representations, the written language, MU5 Autocode, and the parametric compiler target language (CTL).

(Received October 1971)

At an early stage in the design of the MU5 software it was decided to introduce a compiler target language (CTL) into which the high level languages would be translated. For each high level language a *translator* would be provided to convert from the language to CTL while a single compiler converts from CTL to machine code. The objective was to simplify individual translators by forcing the CTL to as high a level as possible. For example, the CTL contains declarations with the characteristics of those found in high level languages so that name and property list management problems are passed to the CTL compiler. This scheme enables the mode of compilation, for example output in semi-compiled form or loading for immediate execution, to be determined within the CTL rather than within each translator.

Subsequently, a further role for the CTL emerged. The MU5 translators could be used on a range of machines provided a CTL compiler could be written for each machine. This machine independence could extend over machines with significant structural differences provided the data and address formats were compatible. This idea is summarised in **Fig. 1.** It is similar to the UNCOL (Strong, Wegstein, Tritter, Olsztyn, Mock, and Steel, 1958) idea except that, whereas UNCOL attempted to span the significant differences between existing machines, the CTL has been designed to suit machines originating from MU5. There is, however, a more significant difference: the communication between the translators and the CTL compiler is two-way. Some of the CTL procedures return information to the translators. For example there is a procedure for interrogating property lists. It is this which allows the whole property and name list organisation to be contained within CTL. The CTL does not have to be encoded in character form by the translators then decoded by the CTL compiler. Instead there is a CTL procedure corresponding to each type of statement, so that the CTL is a body of procedures rather than a written language. The main input parameter of each procedure is a vector whose elements define the nature of the statement. In the case of an arithmetic assignment these elements comprise a sequence of operator operand pairs. Only a small increase in compile time results from using the CTL procedures to generate code, because they form part of a natural progression from source to object code.

A loss of run time efficiency could arise from the translators losing the ability to control completely the code which is generated. This problem is largely irrelevant with MU5 because of the high level nature of the order code. For example, the addressable registers serve dedicated functions which correspond to identifiable features of the high level languages. Also the machine dynamically optimises the use of the fast operand store (Ibbett, 1971). If CTL were to be implemented on machines considerably different from MU5 in these respects then some theoretical inefficiency might result. In practice it is difficult to obtain compilers which compile optimum code, so that the inefficiency may be no worse than that already tolerated on many machines.

In the overall software structure the CTL is the instruction set of the MU5 virtual machine (Morris, Detlefsen, Frank, and Sweeney, 1971). Hence compatibility in the notional MU5 range of machines is at the CTL rather than the order code level. There is an associated written form of CTL, MU5 Autocode, which is the lowest level of programming language and which is used for system programs.

Design considerations

Two principal decisions have determined the overall characteristics of the CTL and the Autocode. The first of these was that the CTL and the Autocode should be structurally the same language. It is thus possible for the CTL compiler to generate the Autocode equivalent of a program in any source language. A number of minor advantages stem from this ranging from the debugging of compilers to the hand optimisation of important programs. In the light of past experience it was also considered advantageous for the compilers to be written in the same language as they generate.

The second decision was that the CTL and the Autocode should be a high level representation of the MU5 machine code. For example, it will be seen that the declarations relate to physical data items in the machine rather than logical data types. Also the variables are typeless, as are operands in the machine, permitting arbitrary manipulation using any kind of arithmetic. Consequently, in MU5 Autocode information about data structures is embedded in the code rather than just in the declarations as in PL/1 or ALGOL 68. However, it is not clear that, on balance, any significant loss of clarity results from this, particularly since operand accessing in MU5 is very flexible. Furthermore, efficiency considerations will often dictate that such structures be carefully designed to fit the

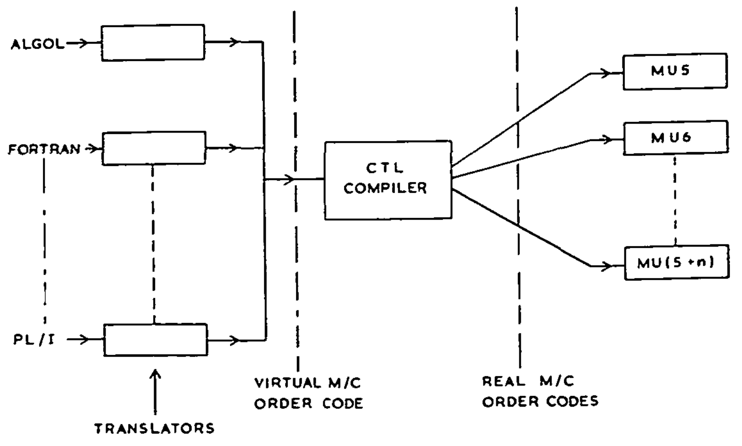


Fig. 1. Compiler Target Language

```
PROC SORT(A,N)
PROC SPEC SUB.OF.MAX(S,I32,I32)I32
V32,P,SUB
V64,DUMP
              PROC SUB.OF.MAX(A,P,N)
              V32,SUB,I
              I32,P => SUB
              CYCLE I = P+1,1,N
              IF[R64,A[I]>A[SUB]]THEN
              I32,I => SUB
              CONTINUE
              REPEAT
              RESULT = SUB
              END
CYCLE P = 1,1,N-1
I32,SUB.OF.MAX(A,P,N) => SUB
R64,A[SUB] => DUMP
R64,A[P] => A[SUB]
R64,DUMP => A[P]
REPEAT
RETURN
END
```

Fig. 2. An example of an MU5 Autocode procedure

machine. Additional practical considerations reinforced this decision. Firstly, because the hardware and software of MU5 will be commissioned together, it was considered preferable for the language to reflect the hardware accurately. Secondly, the dependence of the rest of the software on the CTL and the Autocode necessitates a short time scale for their development.

The Autocode representation is the best way of describing the structure of both this and the CTL. An example of a procedure for sorting an array in descending order using linear selection is given in Fig. 2. From this the basic language structure should be apparent. In the following sections the form of data, the operations available and the overall control structure are described.

The Autocode computation statements

Each arithmetic computation to be performed requires an implicit or explicit specification of the type and size of arithmetic required. The Autocode provides many arithmetic modes but no particular one is considered to be the fundamental mode. It is assumed that only those modes justified by the primary use of a machine are provided in hardware, the rest being provided by software. The arithmetic modes are signed and unsigned integer, real and decimal of size 32, 64 or 128 bits and a Boolean mode. In MU5 32-bit signed and unsigned integer, 32- and 64-bit real, and Boolean modes are provided in hardware together with some special functions to aid the software implementation of other modes. The mode is specified at the start of each statement and is following mainly by operator operand pairs. Each of these pairs generally corresponds to a machine instruction; hence the code compiled is closely controlled.

The operator precedence is strictly left to right, in contrast to most high-level languages. There are several reasons for this. Firstly, the calculations in systems programs are often of a logical rather than a mathematical nature, and use operators for which precedence rules are not well established. Secondly, it is easier to see that efficient code is being compiled when evaluation is left to right than when implicit stacking of partial results is taking place. Thirdly, since different languages have varying precedence rules an equal precedence convention is the most convenient for use in the target language. Precedence can be forced by the use of bracketed sub-expressions which explicitly demand the stacking of a partial result on the opening bracket, and the application of a reverse operation on the closing bracket. This is shown in the following example of a typical statement equivalent to the ALGOL

$$E := (A + B)/(C + D)$$
$$\text{R64}, A + B/(C + D) => E$$

R64 is the 64-bit real mode of calculation; A, B, C, D and E are operands, and / + and => are the divide, add and store operators respectively. In MU5 this statement would translate to:

$ACC = A$	::set the floating-point accumulator to the value of A
$ACC + B$	:: add the value of B
$ACC *= C$	::stack the partial result and load the value of C
$ACC + D$	::add the value of D
$ACC \varnothing STACK$	::reverse divide by the stacked partial result
$ACC => E$	::store the result in E

Operands and declaratives

The names which are used to represent operands must be declared before use. Thus single pass compilation is possible. The user has close control over the store layout and implicit declarations are not permitted. The scope of the declaratives is organised on a block structure basis. The basic items which may be declared are scalars, vectors and strings. The declaratives specify the operand size which for vector elements, where the widest variation is possible, may be 1, 4, 8, 16, 32, 64 or 128 bits. Scalars are recognised in the machine design and special hardware is provided in MU5 to take advantage of their existence (Ibbett, 1971). Vectors are accessed indirectly by means of descriptors. The descriptor, a stored scalar quantity, specifies the origin, bound and element size for a vector. Vector operands consist of the vector name and a subscript expression of arbitrary complexity. An example of the use of vectors is:

$$\text{R64}, X[I \times N] + Y[J - 2] => Z$$

In MU5 a modifier register, B, is used to evaluate subscripts so this statement translates to:

$B = I$	::compute first subscript in B register
$B \times N$	
$ACC = X[B]$	::load required element of X into accumulator
$B = J$	::compute second subscript
$B - 2$	
$ACC + Y[B]$	
$ACC => Z$	

The Autocode also provides for more complicated data structures such as operands accessed through several levels of descriptors and multidimensional arrays. These cases are always explicitly described rather than following implicit rules. Hence, for example, if X is a vector of vector descriptors, $X[I][J]$ causes element J of the Ith vector to be accessed. Sometimes, the address of a data item, rather than its value, is required. In this case, the built-in function 'ADDR' is used. The use of a simple operand, ADDR, or subscripting enables operands to be manipulated in any way required. Also new operands may be created by combining or partitioning existing data structures using the facilities provided for manipulating descriptors.

The allocation of store for these data structures may be dynamic or static. In the latter case store allocation is controlled by declared areas. An example of a static vector declaration is:

VEC/$*AREA* [64, 100] *A*

This declares a vector with 100 64-bit elements numbered 0-99 in the store area *AREA*. A descriptor of the vector is placed in the local namespace of the current procedure and may be referred to as *A*.

Autocode control statements

The order of execution of statements in a program is determined by various control statements. These are intended to encompass the corresponding features of standard high-level languages. Apart from readability requirements, the possibility of a variety of control hardware in different machines forced these statements to a high level.

A Boolean facility similar to that of ALGOL 60 is provided. This requirement is catered for at the hardware level in MU5 partly because the cost was small and partly because of local interest in non-numeric programming. The general form of the conditional statement, and the conditional expression, is also similar to that of ALGOL 60.

A relatively restricted looping facility is provided. Because there are significant structural differences in the various high-level languages it is expected that compilers will usually generate the corresponding conditional statements. The simple facility provided deals with the frequently occurring cases for which special hardware, such as test and count instructions, might exist.

Procedures

A principal design aim of MU5 has been to incorporate the concept of recursive procedures at the hardware level (Kilburn, Morris, Rohl, and Sumner, 1969). It therefore follows that the Autocode includes this facility in a form which reflects those in existing high-level languages.

Therefore procedures have static or dynamic namespaces and parameters which are expressions, corresponding to ALGOL call by value parameters, or descriptors. Descriptor parameters enable reference, substitution, procedure and label parameters to be simply programmed. Procedures which are used as functions yield a result handed back in the accumulator, in which case they may be called in the course of evaluating an expression.

In Fig. 2 it can be seen that a procedure is preceded by a specification. This specification gives the mode of each parameter and of any result yielded by the procedure, while the procedure heading gives only the formal parameter names. Further, the specification must be given before the first call. Thus, the compilation of procedure calls is simplified because the parameters' modes are known.

An example of the parametric form of CTL

An example is now given of the way that this written language is parameterised to form the CTL. Suppose that an ALGOL translator wishes to translate:

$$x := y + 10$$

where x and y are declared integer.

The corresponding Autocode statement is:

$$\text{I32}, y + 10 => x$$

The translator must do two things to process this statement:

1. Assemble a parametric form of the statement into a vector.
2. Call the *CTL.COMPUTATION* procedure with the vector as parameter.

This procedure then generates the corresponding MU5 binary instructions, semi-compiled or other forms.

Suppose that the translator is assembling the parametric form into a vector *CODE* declared:

VEC [32, 21] *CODE*

then the elements of code are assigned as follows:

CODE	[0]	: Computation is in I32 mode and next operand is a name
	[1]	: Name y
	[2]	: Operator +, next operand is a constant
	[3]	: Constant 10
	[4]	: Operator =>, next operand is a name
	[5]	: Name x
	[6]	: Terminating mark

The vector therefore contains an operator operand sequence. Each word containing an operator also describes the type of the operand following. The operator is held in the top 16 bits and the operand type in the bottom 16. Considering, in the preceding example, one such element of *CODE* in more detail, since => is operator 9 and a name is an operand type 16, *CODE*[4] = %00090010 in hexadecimal.

A name is replaced at the lexical analysis stage by an internal identifier, an integer, which is handed back to the translator by the *CTL.ADD.NAME* procedure. Such integers are placed in *CODE* [1] and *CODE* [5]. This form of operand assumes the use of the standard form of name and property lists mentioned previously.

CODE [0] which is specially coded to indicate the mode of the sequence can be regarded as describing a load operation. The complete hexadecimal representation of the previous example is:

CODE	[0]	: %80150010
	[1]	: Integer corresponding to y (internal identifier)
	[2]	: %00012001
	[3]	: %0000000A
	[4]	: %00090010
	[5]	: Internal identifier corresponding to x
	[6]	: %00320000

When this information has been assembled a call:

CTL.COMPUTATION (*CODE*)

can be made, and the CTL procedure generates the code.

Conclusion

The Autocode and CTL have been implemented in a simulated MU5 system on an ICL 1905E. One translator, for Atlas Autocode, is already running in this system. The development of others for ALGOL, FORTRAN and PL/1 is well advanced. The MU5 implementation awaits the commissioning of the hardware after which the translators should be transferred without modification.

A compiler for a subset of the Autocode which generates 1900 code is also available. This is being used to develop operating system modules which will also be transferred to MU5. The 1900 code generated is sufficiently good for these modules to be used as part of the 1905E operating system (Morris, Frank, Robinson, and Wiles, 1971).

References

Ibbett, R. N. (1971). The MU5 Instruction Pipeline, *The Computer Journal*, Vol. 15, No. 1, pp. 43-51.

Kilburn, T., Morris, D., Rohl, J. S., and Sumner, F. H. (1969). A system design proposal, *Information Processing 68*, North Holland Publishing Co., Amsterdam, pp. 806-811.

Morris, D., Detlefsen, G. D., Frank, G. R., and Sweeney, T. J. (1971). The structure of the MU5 operating system, *The Computer Journal*, Vol. 15, No. 2, pp. 113-115.

Morris, D., Frank, G. R., Robinson, P. H., and Wiles, P. R. (1972). The supervisors of the MU5 operating system (to be published).

Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T. (1958). The problem of programming communication with changing machines, *CACM*, Vol. 1, No. 8, pp. 12-18.

Correspondence

To the Editor
The Computer Journal

Sir

The recent paper by L. B. Smith ('Drawing Ellipses, . . .', Vol. 14, No. 1) suggests that a good criterion for approximating a convex curve by an inscribed polygon with a fixed number of vertices is that the N-gon have maximal area. In the case of conic sections this criterion leads to highly efficient algorithms, as Mr. Smith so clearly illustrates.

The paper gives a Lemma which states that an N-gon inscribed into a convex curve has maximal area if and only if the tangent to the curve at each vertex of the N-gon is parallel to the chord determined by the two adjacent vertices. The justification of the Lemma considered only the possibility of moving just one vertex at a time, so it is not surprising to find counter-examples which necessitate moving two or more vertices of the polygon. The triangle joining the midpoints of the sides of any given triangle satisfies the conditions of the Lemma, yet it is not maximal. The area of the inscribed triangle cannot be increased by altering any one of its vertices. In the following discussion an inscribed polygon such that the slope at each of its vertices is parallel to the chord of its two neighbouring vertices will be called a *locally maximum* polygon.

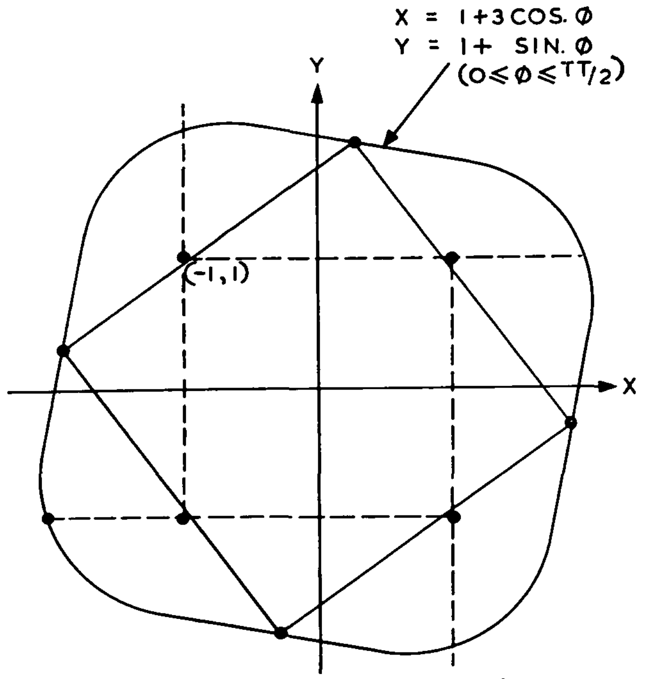
A locally maximum N-gon for a given arbitrary convex curve can be determined iteratively starting from an initial set of N points on the curve by moving one point after another to a point further away (if possible) from the chord of its two neighbouring vertices. Convergence is guaranteed and fairly rapid.

Some curves have but one locally maximum N-gon, in which case the conditions of the Lemma prove sufficient. The curve defined by $x = -1 + 3\cos\theta$, $y = 1 + \sin\theta$, for $0 \leqslant \theta \leqslant \pi/2$, and its rotations about the origin into the other quadrants, is one such example. It has but one locally maximum quadrilateral, with a vertex in each quadrant corresponding to the value of θ approximately 64°7′, as shown in **Fig. 1**.

The curve $x^4 + y^4 = 1$ has just two locally maximum octagons. One of them has vertices $(\pm 1, 0)$, $(0, \pm 1)$, and the points $(\pm c, \pm c)$, where $c^4 = 1$. This octagon, of area $4c$, is 'unstable', in the sense that if any one of its vertices is shifted slightly, then either one of its two neighbours can be shifted slightly to increase the area to *more* than $4c$. This means that if the vertices are shifted successively so as to increase the area at each step, they must ultimately approach the only other locally maximum octagon, namely the points $(\pm x, \pm y)$ and $(\pm y, \pm x)$, where $x^4 = (3 + \sqrt{5})/6$ and $y^4 = (3 - \sqrt{5})/6$. This octagon has area $2\sqrt{3}$. Once again, the lack of sufficiency of the conditions of the Lemma do not interfere with finding a maximal quadrilateral. The curve is shown in **Fig. 2**.

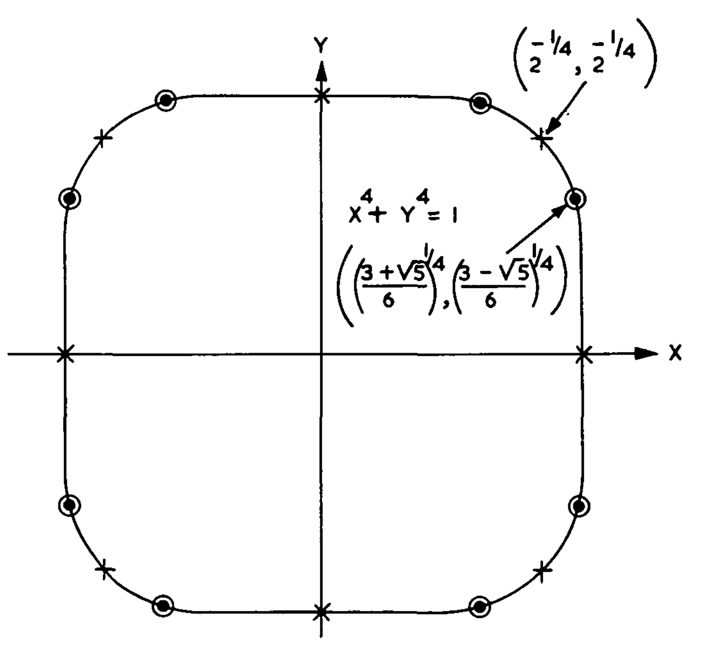
It is possible for a convex curve to possess many distinct maximal N-gons, with none of them unstable. Consider the convex curve determined by the two parabolas, $y^2 = 16 + 64x$ and $y^2 = 16 - x$, which intersect at $(0, \pm 4)$. A locally maximum 65-gon may have from 5 to 65 vertices on $y^2 = 16 - x$, the remaining vertices lying on $y^2 = 16 + 64x$. The points on either parabola must have ordinates distributed equally from -4 to $+4$. The more vertices on

Continued on page 116



THIS CURVE HAS BUT ONE LOCALLY MAXIMUM QUADRILATERAL. DETERMINED BY Θ≈64°7′ IN THE FIRST QUADRANT AND 3 ROTATIONS OF IT.

Fig. 1


Y
$(2^{-1/4}, 2^{-1/4})$
$X^4 + Y^4 = 1$
$\left(\left(\frac{3+\sqrt{5}}{6}\right)^{1/4}, \left(\frac{3-\sqrt{5}}{6}\right)^{1/4}\right)$
X

THE 8 POINTS MARKED ◉ FORM A MAXIMAL OCTAGON.
THE 8 MARKED X FORM AN UNSTABLE OCTAGON.

Fig. 2