A NEW APPROACH TO THE FUNCTIONAL DESIGN OF A DIGITAL COMPUTER

R. S. Barton
Computer Consultant
Altadena, California

Summary

The present methods of determining the functional design of computers are critically reviewed and a new approach proposed. This is illustrated by explaining, in abstracted form, part of the control organization of a new and different machine based, in part, on the ALGOL 60 language.[1] The concepts of expression and procedure lead directly to use of a Polish string program. A new arrangement of control registers results, which provides for automatic allocation of temporary storage within expressions and procedures, and a generalized subroutine linkage.

The simplicity and power of these notions suggests that there is much room for improvement in present machines and that more attention should be given to control functions in new designs.

Introduction

The ideas presented arise from the conviction that for a true general purpose digital computer both coding and operation should be fully automated. Higher level programming languages, such as ALGOL, should be employed to the practical exclusion of machine language; questions of efficiency of object program and translation process ought not to arise if the machine has been properly designed. Operation should be under the control of the machine itself, in a fuller sense than is typical in current practice. The functions of scheduling, segmentation of programs for multi-level storage, and control of input-output operations should be handled by a general operational program.

This new approach will be illustrated after reviewing the customary methods of machine design.

The Special Purpose Machine

In simple and well defined applications, the design engineer may dispense entirely with programming assistance and the program may be entirely, or in large part, in the hardware. If the processing required is complex, programmers are invited to assist the engineers. There will be a period of trading-off programmed and component logic, but the resulting machine will tend to resemble the conventional general purpose computer.

The Engineers' General Purpose Machine

In the design of machines to meet competition, the utilization of new components is likely to be of vital concern to the designers. While requiring a complete new set of programs, the new product seldom shows more than minor variations of the traditional design. Its new features originate with both programmers and logical designers, but those ideas which are contributed by the programmers stem usually from applications experience with previous machines rather than from systematic theory. The logical designer has the last word and is most likely to accept ideas which require a minimum of new design.

The Programmers' General Purpose Machine

It seems to be the case that as yet no machine's design has been significantly effected by persons experienced in the development of automatic programming systems. Programming still must be imposed upon designs that have been determined by marketing pressures and tradition. It must be admitted that the programmers of the last decade have been poorly prepared to make the kind of contribution that should be expected. The art of programming has developed in a helter-skelter manner, leaving behind little of value. There is almost no theory and little standard methodology. The logical designers have a large body of switching theory as a basis for their work and have, consequently, done better.

The Simultaneous Design of Computers and Programming Systems

Rather than hope for a new spirit of cooperation among disparate product planning, engineering, and programming departments, a single small organization is needed for each product conception. This would be comprised of three kinds of people responsible for aspects of system model, program design, and logical design.

As an example, for a computer to process applications expressible in the ALGOL 60 language, the system model group would interpret the language, specify a hardware representation and necessary language supplements, define speed or cost objectives, and the "use image" the machine is to present. They would have responsibility for ensuring proper interpretation of the model by both programmers and logical designers.

The program designers would have background experience in the construction of translators and some knowledge of logical design. They will consider necessary reductions in the source language and translation techniques to enable efficient object time interpretation by hardware.

The logical designers would be oriented in current programming practice and become familiar with the source programming language to be used. Their task is to produce designs to handle the reduced languages.

Concepts for the Design

Some simple ideas are now presented that arise quite naturally from using the ALGOL 60 language as a model. These ideas have actually entered into the design of a new Burroughs Information Processing System. For purposes of exposition, they will be considered out of the context of the actual machine, and liberties will be taken to avoid complications which are not germane to the subject.

Polish String Program

The Polish notation was invented by the logician Lukasiewicz for use in the propositional calculus. It has the advantage that rules of operator precedence and signs of grouping are not required. At least two logical machines have been built which use it as a source language. The first of these was the Burroughs Truth Function Evaluator[2]. More recently, Bauer described a similar logical device[3] and hinted of remarkable results which were discussed in another paper[4]. During the past few years, numerous persons have "discovered" the extension of this notation to arithmetic expressions and found it useful as an intermediate language in designing algebraic translators.

Any expression, as defined by ALGOL 60, can be simply translated into Polish form, and this can be the basis of an efficient machine language to be used in place of the customary single or multiple-address command formats. Methods for doing this are well known and familiarity with an algorithm is assumed. While it would be quite feasible to construct a machine to directly interpret ALGOL expressions having suitably restricted identifiers, it was decided, in view of the simplicity of the transformation, to use the Polish form. At this point, it is important to stress that programs need never be written in Polish form; and no lower level assembly-type language is required.

This form provides a machine language with many desirable properties. Programs consist of a string of elements which correspond to identifiers, literals, or operators. All the operators defined in ALGOL 60 can be included. Using the stack construct, next described, there is no need for store and fetch commands such as are normally associated with the temporary storage of intermediate quantities in the evaluation of expressions. More important, the resulting program is in a better form for the application of procedures to improve program efficiency than is the case with the usual command languages.

The Stack Construct

Assume that the elements of the string are examined from left to right so that operators need not be deferred. Normally, for unary and binary operators, the transformation to Polish form would ensure (excepting in the case of procedures) that at most two operands would have to be fetched before each execution.

To mechanize the Polish string program, a special address-length register called the stack counter is provided to hold the most current cell address in a vector of temporary storage cells referred to as the stack. Two word-length arithmetic registers hold the most recently fetched or computed operands. Associated with each of the latter is an occupancy toggle for indicating whether or not that register contains an operand.

The action of the stack is defined in Table I. The following notation applies: S, A, B, T_A, T_B represent the contents of the stack counter, arithmetic registers, and occupancy toggles. X is an operand. $\ominus$ and $\oplus$ are unary and binary operators. S* signifies the contents of the cell addressed by the contents of S.

Now, while the case of stack operation, which is presented is somewhat simplified, it does show how a Polish string program can be mechanized. The occurrences of operators with $T_A = T_B = 0$ is actually abnormal, but are included to allow continuation in the event that the evaluation of an expression is interrupted. The state $T_A = 1$, $T_B = 0$ is unstable.

It is worth noting that upon completion of the evaluation of a well-formed expression we have $T_A = 0$, $T_B = 1$, with the value of S upon completion equal to its initial value. This offers a possibility for checking in the hardware which does not exist in such a simple form for the usual command language.

In the event that the reader does not happen to be familiar with the Lukasiewicz notation, he may find it useful to trace the operation of the stack for the program XY + VW/U + x Z + which corresponds to the expression (X + Y) x (U + V/W) + Z.

Subroutines and ALGOL Procedures

To realize additional important advantages from this program format, we extend these notions to handle n-ary operators or n-place functions that are defined by sub-programs. The important case of call-by-name is deferred. Call-by-value only is discussed. Declarations of functions will cause subroutines to be generated and extensions of the operator set to be defined. Similarly, it is assumed that for each array one or more storage-mapping functions have been defined and that, corresponding to each, a subroutine has been created from the array declaration which will have access to information on bounds of indices and the base address of the array. Such subroutines will also be called by extensions of the operator set. The program corresponding to the array element or function call will then consist of an ordered set of expressions representing the indices or arguments. The values of these are entered into the stack. When the operator corresponding to that subroutine is encountered, linkage automatically results.

Subroutine Control Using the Stack

It is now necessary to place the contents of registers A and B (or B if A is empty) into the stack since the subroutine to be executed will generally require these registers. Furthermore, the contents of the control counter, C, must be saved to enable return from the subroutine. This return can be saved in the stack and the position of this address recorded in another address-length register designated F. To afford a link to the return for a possible higher level subroutine, the former contents of F are retained in the cell with C. Finally, the subroutine entry address specified by the extension operator is entered into C and linkage to the subroutine is complete.

Within the subroutine, the parameters are addressed minus relative to F in reverse order. Temporary storage is allocated for the subroutine by advancing S corresponding to the number of cells needed. These cells can then be addressed plus relative to F. It will also be useful to store constants used by the subroutine ahead of its entry and address them minus relative to C.

Upon exit, the resulting value is left in B, the contents of S are replaced by F, and for the cell then addressed by S, the C-part will go to C and the F-part to F. Finally, S is reduced by n-1. This automatic linkage construct enables the use of subroutines in depth and a subroutine may call itself.

Denoting the F and C parts of S* by S^*_F and S^*_C, respectively, the location of the subroutine by P, and other notation as previously defined, the action upon entry and exit is displayed in Table II.

Let it now be assumed that the execution of a scanned program element is delayed until the following syllable has been interpreted. Operators can be defined which force the preceding syllable to be a call-by-name. Each word in the stack has a control bit that distinguishes between values and names. The address referenced by the element preceding the call-by-name operator then is entered into the stack and the control bit for that cell set to indicate a name. If an operator is encountered, followed by the call-by-name operator, the operator (or the location of the subroutine which effects execution of an extended operator) goes into the stack and the control bit is set. Within a subroutine, any reference to this stack cell that is not followed by a call-by-name operator will cause execution. If, otherwise, the reference to the cell is again followed by a call-by-name operator, the cell contents are copied into the new stack level. A similar action results whenever a parameter of one subroutine is used in a lower level subroutine.

Other Consequences of the Design

Some of the key aspects of the logical organization of the machine have been introduced in a gradual fashion. While not all of the consequences of the model which have been developed can be presented in this paper, a few concluding remarks may be of value.

Change of sequence is accomplished by one of two means: jumps relative to C of conditional or unconditional type, or via a switching table of entries corresponding to labeled program segments. The conditional jumps examine the truth value in the stack produced by evaluation of a Boolean expression, and then cause it to be erased. For ease of segmentation and effective use of a random-access secondary storage device, we make program invariant with respect to its position in storage. Corresponding to each declaration of array, switch, or procedure will be a "locator" word which is assigned in a table. Program references are then made to these words to obtain the location of the corresponding program. These words contain other information which is itilized in multi-programming and automatic segmentation control.

"Locator" words also correspond to labeled program segments and input-output control information. The latter are grouped for scanning by a universal input-output control program which assigns I/O channels. The main program and I/O control program communicate via a status bit in these words.

Character and bit manipulation constructs for the machine are also departures from familiar practice, but arise from different considerations and will not be discussed here.

Conclusion

It is hoped that a case has been made for a way to introduce some new ideas into a field where enormous amounts of technical talent are spent in designing the hardware and programs for a large number of very similar machines. Most of these are "general purpose." Much of the effort in developing these machines could better be spent in designing some useful "special purpose" computers. An ALGOL machine would fit into the latter category. With automated logical design and fabrication in the immediate future, any number of these useful special purpose machines can be envisaged.

* * * *

This paper is based on work sponsored by the Burroughs Corporation.

References

1. Naur, P. (Editor): Report on the Algorithmic Language ALGOL 60, Comm. Assn. Comp. Mach. 3, No. 5 (1960), 299-314.

2. Burks, A.W., Warren, D.W., Wright, J. B.: An Analysis of a Logical Machine Using Parentheses-Free Notation, Math. Tables Aids Comp. 9 (1954), 53-57.

3. Bauer, F. L.: The Formula-Controlled Logical Computer "Stanislaus," Math. Comp. 14, No. 69 (1960), 64-67.

4. Samelson, K. and Bauer, F. L.: Sequential Formula Translation, Comm. Assn. Comp. Mach. 3, No. 2 (1960), 76-83.

	OPERAND	UNARY OPERATOR	BINARY OPERATOR
$T_A = T_B = 0$	1. $X \rightarrow A$, $T_A = 1$ 2. $A \rightarrow B$, $T_B = 1$, $T_A = 0$	1. $S^* \rightarrow B$, $T_B = 1$ 2. $S-1 \rightarrow S$ 3. $B\Theta \rightarrow B$	1. $S^* \rightarrow B$, $T_B = 1$ 2. $S-1 \rightarrow S$ 3. $B \rightarrow A$, $T_B = 0$, $T_A = 1$ 4. $S^* \rightarrow B$, $T_B = 1$ 5. $S-1 \rightarrow S$ 6. $AB\oplus \rightarrow B$, $T_A = 0$
$T_A = 0$ $T_B = 1$	1. $X \rightarrow A$, $T_A = 1$	1. $B\Theta \rightarrow B$	3. $B \rightarrow A$, $T_B = 0$, $T_A = 1$ 4. $S^* \rightarrow B$, $T_B = 1$ 5. $S-1 \rightarrow S$ 6. $AB\oplus \rightarrow B$, $T_A = 0$
$T_A = T_B = 1$	1. $S+1 \rightarrow S$ 2. $B \rightarrow S^*$ 3. $A \rightarrow B$ 4. $X \rightarrow A$	1. $A\Theta \rightarrow A$	6. $AB\oplus \rightarrow B$, $T_A = 0$

TABLE I

	ENTRY	EXIT
$T_A = T_B = 0$	6. $S+1 \rightarrow S$ 7. $F \rightarrow S^*_F$, $C \rightarrow S^*_C$ 8. $S \rightarrow F$ 9. $P \rightarrow C$	(Normal for procedure without value) 1. $F \rightarrow S$ 2. $S^*_F \rightarrow F$, $S^*_C \rightarrow C$ 3. $S-n-1 \rightarrow S$
$T_A = 0$ $T_B = 1$	4. $S+1 \rightarrow S$ 5. $B \rightarrow S^*$, $T_B = 0$ Steps 6 through 9	(Normal for procedure with value) (Same as above)
$T_A = T_B = 1$	1. $S+1 \rightarrow S$ 2. $B \rightarrow S^*$ 3. $A \rightarrow B$, $T_A = 0$ 4. $S+1 \rightarrow S$ 5. $B \rightarrow S^*$, $T_B = 0$ Steps 6 through 9	(Improper)

TABLE II