N. WIRTH, Editor

A Microprogrammed Implementation of EULER on IBM System/360 Model 30

HELMUT WEBER
International Business Machines Corp., Endicott, N. Y.*

An experimental processing system for the algorithmic language EULER has been implemented in microprogramming on an IBM System/360 Model 30 using a second Read-Only Storage unit. The system consists of a microprogrammed compiler and a microprogrammed String Language Interpreter, and of an I/O control program written in 360 machine language.

The system is described and results are given in terms of microprogram and main storage space required and compiler and interpreter performance obtained. The role of microprogramming is stressed, which opens a new dimension in the processing of interpretive code. The structure and content of a higher level language can be matched by an appropriate interpretive language which can be executed efficiently by microprograms on existing computer hardware.

1. Introduction

Programs written in a procedure-oriented language are usually processed in two steps. They are first translated into an equivalent form which is more efficiently interpretable; then the translated text is interpreted ("executed") by an interpretation mechanism. The translation process is a data-invariant and flow-invariant operation. It consists of two parts—an analytical part, which analyzes the higher level language text, and a generative part, which builds up a string of instructions that can be directly interpreted by a machine. The analytical part of the translator depends on the higher level language; the generative part depends on a set of instructions interpretable by a machine. Historically there was only one set of instructions which could be interpreted efficiently by a machine, its "machine language." Figure 1 outlines this scheme.

Some of the processors of the IBM System/360 family are microprogrammed machines. On them the "360 machine language" is interpreted not by wired-in logic but by an interpretive microprogram, stored in control storage, which in turn is interpreted by wired-in logic. Therefore, in a certain sense the 360 language is not the "machine language" of these processors but the (efficiently interpretable) language in which the processors of the System/360 family are compatible. The true "machine language" of these processors is their microprogram language. This language is on a lower level than the "360 language"; it contains the elementary operations of the machine as operators and the elements of the data flow and storage as operands.

Now it is conceivable to compile a program written in a higher level language into a microprogram language string. This string would undoubtedly contain substrings which occur over and over in the same sequence. We could call these substrings procedures and move them out of the main string, replacing their occurrence by a procedure call symbol, followed by a parameter designator pointing to the particular procedure. Our object program then takes on the appearance of a sequence of call statements. From here it is only a final step to eliminate the call symbols and furnish an interpreting mechanism which interprets the remaining sequence of "procedure designators."

The process just described will result in the definition of a string language and the development of a microprogrammed interpretation system to interpret texts in this string language. The situation is similar to the System/360 case: the string language corresponds to the 360 language. Programs written in a higher level language are compiled into string language text to be stored in main storage. The string language interpreter corresponds to the microprogram which interprets 360 language texts. It consists of a recognizing part to read the next consecutive string element and to branch to an appropriate action routine and of action routines to execute the particular procedure called for by the string element.

The essential difference between our situation and the 360 case is that the string language reflects the features of

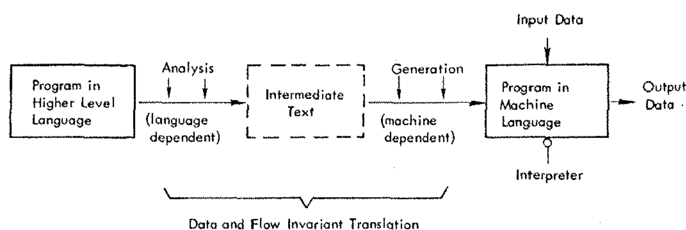


FIG. 1. Processing programs written in higher level languages via translation to machine language

* Systems Development Division

the particular higher level language as well as the features of the particular hardware better than the general purpose 360 language.

What is gained by defining this string language and by providing a microprogrammed interpreter for it? From the method of definition described, it can be seen that the elements of the string language correspond directly to the elements of the higher level language after all simplifying data-invariant and flow-invariant transformations have been performed. But the elements of the string language are also well-adapted to the microprogram structure of the machine. Therefore, during the compiling process (see Figure 2) only a minimum of generation is necessary to produce the string language text. The compiler is shorter and runs faster.

But the more important aspect is that object code execution is also faster. The string language interpreter in case 2 will be coded to take care of all necessary operations in a concise form, whereas in case 1 it will be necessary to compile a whole sequence of machine language instructions for an elementary operation in the higher level language. Examples of this are the compilation of 360 code for an add operation in COBOL of two numbers with different scaling factors or the compilation of machine instructions for table lookup or search operations, etc. In these cases the string language interpreter of Figure 2 will execute a function much faster than the machine language interpreter of Figure 1 will execute the equivalent sequence of machine language instructions. Therefore, object code execution will be faster in scheme 2.

If object code performance is not as much in demand as object storage space economy, the string language interpreter can also be written such that the string language is as tightly packed as possible so that the translated program is as compact as possible and will take up less storage space than the equivalent machine language program under the scheme of Figure 1.

These ideas are applied in an experimental microprogram system for the higher level language EULER [1] described below. Problem areas in this approach are indicated and some ideas for future development are offered.

2. Special Considerations for EULER

The higher level language EULER [1] is a dynamic language. This means that for programs written in it many things have to be done at object code execution time which

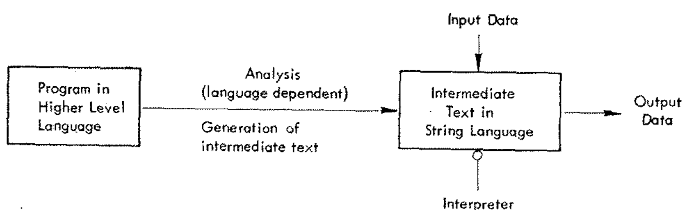


FIG. 2. Processing programs written in higher level languages via translation to interpretive language

can be done at compile time for other languages. EULER also contains basic functions which do not have comparable basic counterparts in the machine languages of most machines. To compile machine code for these dynamic properties and for those special functions would require rather lengthy sequences of machine language instructions, which would consume considerable object code space and require high object code execution time. Therefore, for a language like EULER, interpretation at the string language level by an interpreter into which the dynamic features and special functions are included by microcode will yield much higher object code economy and object code performance than compilation to machine language and interpretation of this machine language.

Three examples from EULER are given here.

1. *Dynamic Type Handling.* To a variable in EULER, constants of varying type can be assigned dynamically. For example in

$A \leftarrow 3;\ \cdots;\ A \leftarrow 4.5_{10}\text{-}5;\ \cdots;\ A \leftarrow$ **true**; $\cdots;\ A \leftarrow$ '$\cdots$';

the quantities assigned to the variable A have the types: integer, real, logical, procedure. Therefore, in EULER each quantity has to carry its type indicator along and each operator operating on a variable has to perform a dynamic type test. The adding operator $+$ for instance in $A + B$ has to test dynamically whether both operands are of type number (integer or real). This type testing is done by the String Language Interpreter in minimum time, whereas it would require extra instructions if the program were to be compiled to 360 machine language.

2. *Recursive Procedures and Dynamic Storage Allocation.* In EULER, procedures can be called recursively, e.g.,

$F \leftarrow$ '**formal** N; **if** $N = 0$ **then** 1 **else** $N{*}F(N - 1)$';

and storage is allocated dynamically, e.g.,

new N; $\cdots$; $N \leftarrow 4$; $\cdots$; **begin new** A; $A \leftarrow$ **list** N;

In order to cope with these problems the EULER execution system uses a run time stack. Each operation is accompanied by stack pointer manipulations which by the microprogram can be accomplished in minimum time (in general, even without extra time because they are overlapped with the operation proper), whereas extra instructions would be required, if the program were compiled.

3. *List Processing.* EULER includes a list processing system, and lists are of a general tree structure, e.g.,

$A \leftarrow (3, 4, (5, 6, 7),$ **true**, '$\cdots$');

List operators are provided like **tail** and **cat** and subscripting:

$B \leftarrow A[3]$; $C \leftarrow B$ **cat** A; $C \leftarrow$ **tail** C;

The string language interpreter handles list operations

directly and efficiently by special microprograms. If the program would be compiled to 360 machine language, a sequence of instructions would be required for each list operation.

3. EULER System on IBM System/360 Model 30

An experimental processing system for the EULER language has been written to demonstrate the validity of these ideas. It is a system running under the IBM Basic Operating System and consists of three parts:

(1) A translator, written in Model 30 microcode.[1] This translator is a one-pass syntax-driven compiler which translates EULER source language programs into a reverse polish string form.

(2) An interpreter, written in Model 30 microcode,[1] which interprets string language programs.

(3) An I/O Control Program written in 360 machine language.[2] This IOCP links the translator and interpreter to the operating system and handles all I/O requests of the translator and interpreter.

The system is an experimental system. Not all the features of EULER are included,—only the general principles that are to be demonstrated. The restrictions are:

(1) Real numbers are not included; only integers are recognized.

(2) The interpreter microprograms for the operators Divide, Integer Divide, Remainder, and Exponentiation have not been coded.

(3) The type 'symbol' is not included.

(4) No garbage collector is provided. Therefore, the system comes to an error stop if a list processing program has used up all available storage space (32K bytes).

Also for reasons of simplicity, the system is written only for a 64K System/360 Model 30 and the storage areas for tables, compiled programs, stacks and free space are assigned fixed addresses.

The string language into which source programs are translated is defined as closely as possible to the interpretive language used in the definition of EULER [1]. The question whether this is the ideal directly interpretable language corresponding to the EULER source language given the Model 30 hardware is left open. Also no attempt is made to define the string language so that it becomes relocatable for use in time sharing or conversational processing mode.

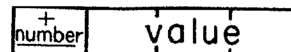
The three storage areas used by the execution system are:

(1) Program Area

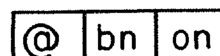
(2) Stack

(3) Variable Area.

Program Area. A translated program in string language consists of a sequence of one-byte symbols for the operators (+, −, **begin, end,** ←, **go to,** etc.). Some of the symbols have trailer bytes associated with them; for instance, the symbol +*number* has three trailer bytes for a 24-bit absolute value of the integer constant.



The symbol *reference* (@) has two trailer bytes, one containing the block number (*bn*), the second one the ordinal number (*on*).



The operators **then, else, and, or** and ' have two trailer bytes containing a 16-bit absolute program address, e.g.,

Other operators with trailer bytes are **label** and the list-building operator.

Stack. The execution time stack consists of a sequence of 32-bit words. It contains block and procedure marks to control the processing of blocks and procedures and temporary values of the various types. The first 4-bit digit of a word in stack always is a type indicator. The format of these words is given in Figure 3.

Variable Area. The variable area is an area (32K bytes long) of 32-bit words used for the storage of values assigned to variables and lists (and also for auxiliary words in procedure descriptors; see type procedure in Figure 3). The format of the entries is exactly the same as the format of the stack entries (see Figure 3), the only exception being that a mark can never occur in the variable area.

4. Microprogramming the IBM System/360 Model 30 [2]

Microprograms are sequences of microprogram words. A microprogram word is composed of 60 bits and contains various fields which control the basic functions in the IBM System/360 Model 30 CPU. These basic functions are storage control, control of the data flow registers and the Arithmetic-Logic-Unit (ALU), microprogram sequencing and branching control, and status bit-setting control. Microprogram words are stored in a Card Capacitor Read-Only Storage (CCROS). Fetching one microprogram word and executing it takes 750 nsec, the basic machine cycle.

[1] Stored in the second Read-Only Storage (Compatibility ROS) of Model 30.

[2] The 360 microprograms are stored in the first Read-Only Storage (360 ROS) of the Model 30.

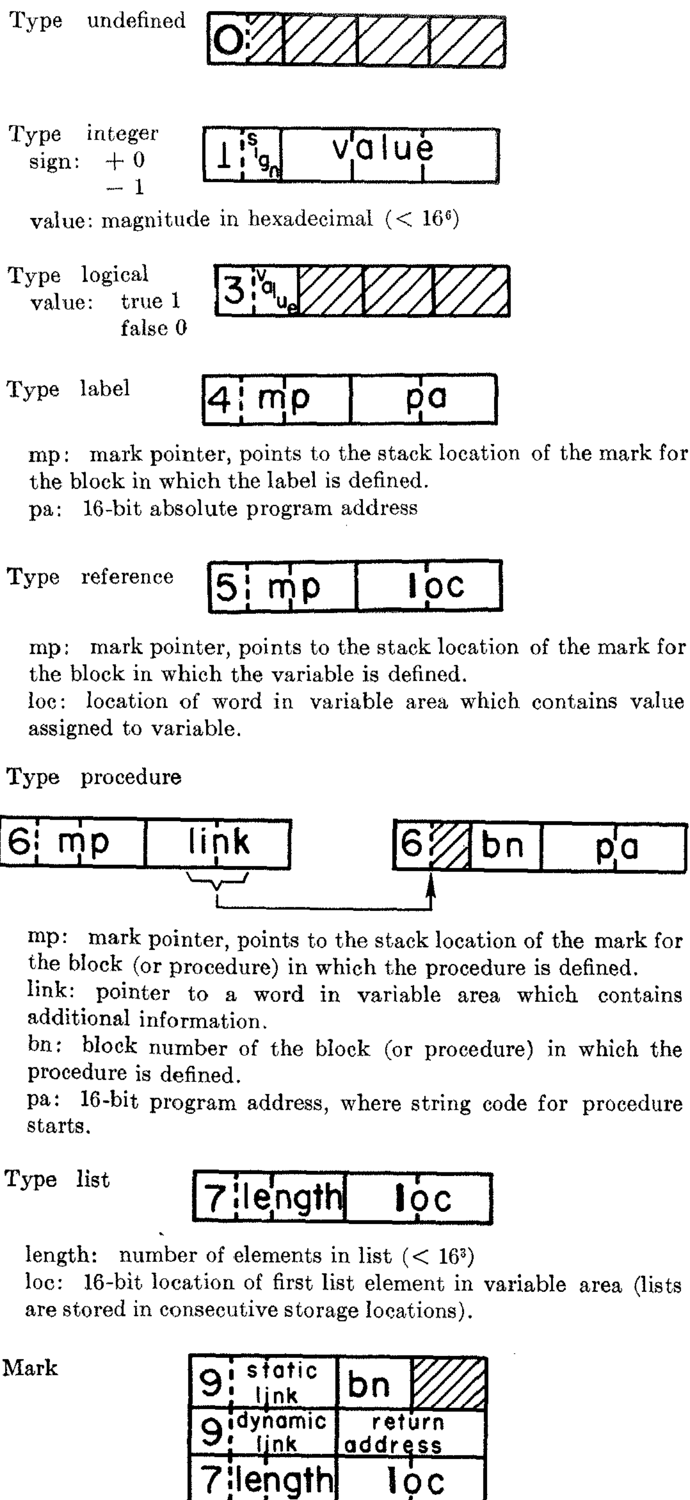


FIG. 3. Format of words in stack and variable area

Figure 4 shows in simplified form the data flow of the IBM System/360 (IBM 2030 CPU). It consists of a core storage with up to 65,536 8-bit bytes and a local storage (accessible by the microprogrammer but not explicitly by the 360 language programmer), a 16-bit storage address register (M, N), a set of 10 8-bit data registers (I, J, $\cdots$, R), an arithmetic-logic-unit (ALU), connecting 8-bit wide buses (Z, A, B, M, N-bus), temporary registers (A, B), switches and gates.

Figure 5 shows the more important fields of a microprogram word. Only 47 bits are shown. Other fields contain various parity bits and special control bits. The field interpretation given in Figure 5 is as for microprogram words in the second Read-Only Storage unit (Compatibility ROS) if the machine is equipped with the 1620 Compatibility Feature. The meaning of the microprogram word fields is explained in connection with Figure 6 which shows the symbolic representation of a microprogram word together with an example as it appears on a microprogram documentation sheet.

The fields of the microprogram word can be grouped in five categories:

1. ALU control fields: CA, CF, CB, CG, CV, CD, CC
2. Storage control fields: CM, CU
3. Microprogram sequencing and branching fields: CN, CH, CL
4. Status bit setting field: CS
5. Constant field: CK

ALU Control Fields. On the line designated "ALU" in Figure 6, an ALU statement can appear. It will specify an A-source and a B-source, possibly an A-source modifier and a B-source modifier, an operator, a destination, and possibly a carry-in control and a carry-out control.

CA is the A-source field. It controls which one of the 10 8-bit data registers is connected to the transient A-register and therefore to the A-input of the ALU.

CB is the B-source field. It controls whether the R, L, or D-register or the CK-field is connected to the transient B-register and therefore to the B-input of the ALU. If "K" (CB=3) is specified in this field, the 4-bit constant field CK is doubled up; i.e., the same four bits are used as the high digit and the low digit.

Between the A-register and the ALU input is a straight/cross switch and a high/low gate. Its function is controlled by the CF-field. Depending on the value of this field, no input is gated into the ALU (0) or only the low (L) or high digit (H) is admitted. CF = 3 gates all eight bits straight through, whereas the codes CF = 5, 6, and 7 cross over the two digits of the byte before admitting the low (XL) or high digit (XH) or both digits (X).

Between the B-register and the ALU input is a high/low gate and a true/complement control. The high/low gate is controlled by the CG-field in the same manner as the high/low gate in the A-input. The true/complement control is operated by the CV-field. It admits the true byte to

the ALU (+) or the inverted byte (−) or controls a six-correct mechanism for decimal addition (@).

The operator and carry controls are given by the CC-field. This field specifies binary addition without carry handling (+0), addition with injection of a 1 (+1) (for instance, to simulate subtraction in connection with the B-input inverter), addition with saving the carry in bit 3 of register S (+0,Save C, and +1,Save C), and addition

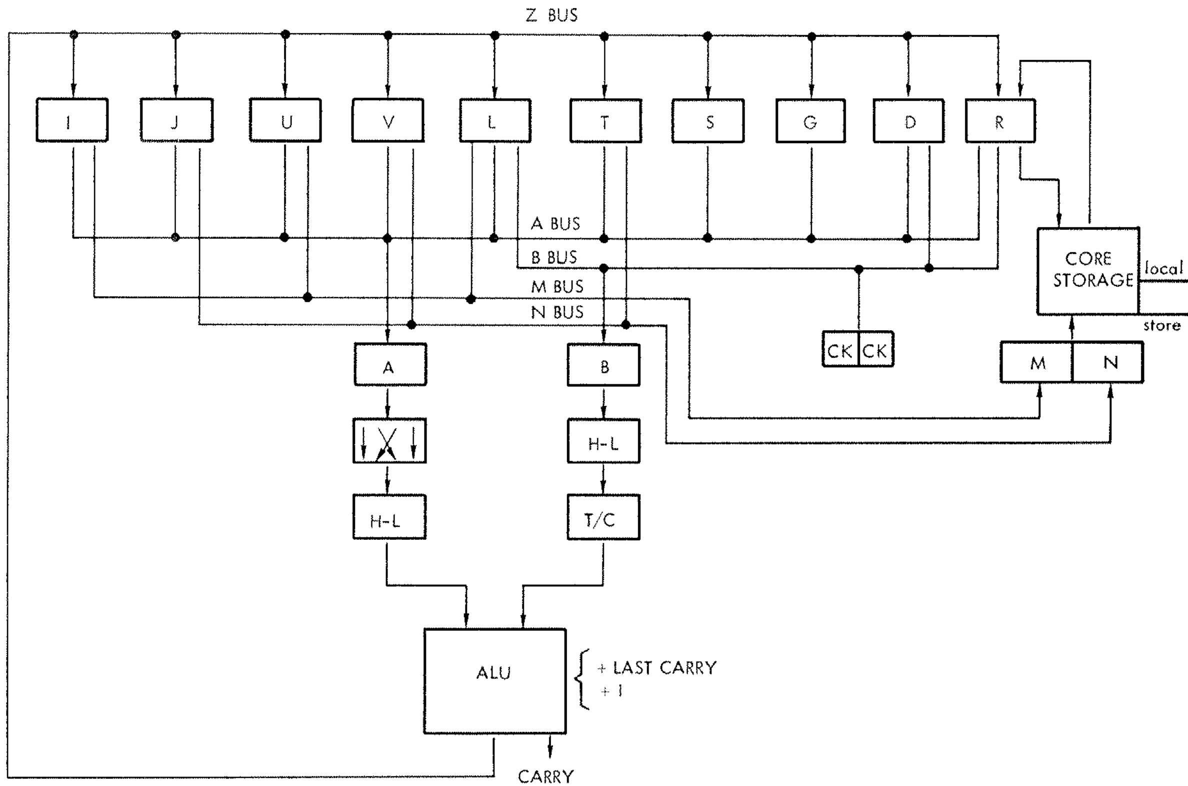
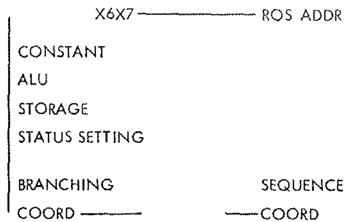


Fig. 4. Simplified data flow of the IBM System/360 Model 30

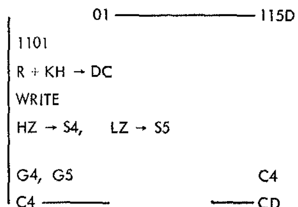
	CN	CH	CL	CM	CU	CA	CB	CK	CD	CF	CG	CV	CC	CS
0000		0	0	WRITE	MS	*	R	0	Ƶ	0	0	+	+0	NO STATUS SETTING
0001		1	1	NOACCESS	LS	*	L	1	*	L	L	−	+1	LƵ→S5
0010		R0	*	STORE	*	*	D	2	*	H	H	*	AND	HƵ→S4
0011		S1	*	IJ→MN	*	*	K	3	*	THROUGH	THR.	@	OR	HƵ→S4,LƵ→S5
0100		*	G1	UV→MN		S		4	*	*			+0,SAVE C	0→S4,0→S5
0101		*	R=VALID DEC	LT→MN		*		5	*	XL			+1,SAVE C	1→S1
0110		ALU CARRY	R1	*		*		6	S	XH			+C,SAVE C	0→S0
0111		S0	Ƶ=0	*		R		7	R	X			XOR	1→S0
1000		R2	G7			D		8	D					0→S2
1001		S2	S3			L		9	L					ANSNƵ→S2
1010		S4	S5			G		X'A'[1]	G					0→S6
1011		S6	S7			T		X'B'	T					1→S6
1100		G0	R3			V		X'C'	V					0→S7
1101		G2	G3			U		X'D'	U					1→S7
1110		G4	G5			J		X'E'	J					*
1111		G6	INTERRUPT			I		X'F'	I					0→S1

[1] X'A' means hexadecimal digit A = 1010

Fig. 5. IBM System 360 Model 30 microprogram word. (Detailed explanation is provided in text.) The field interpretation is given for microprogram words in compatibility ROS if the machine is equipped with the 1620 compatibility feature. Fields marked "*" contain designators not explained here in order not to confuse the basic principles.



Format of Symbolic Representation



Example

Fig. 6. Symbolic representation of a System/360 Model 30 microprogram word

using an old carry stored in bit 3 of register S and saving the new carry in this same bit (+C,Save C). Other codes specify logical operations (AND, OR, XOR).

The CD-field specifies into which register the result of the ALU operation is gated. Any one of the 10 data registers can be specified. Z means that the ALU output is gated nowhere and will be lost.

Storage Control Fields. On the line designated "storage" in Figure 6, a storage statement can appear. It will specify whether this microcycle is a read cycle, a write cycle, a store cycle or a no-storage access cycle, and from where the storage address is supplied (CM-field) and whether storage access is to main storage or local storage (CU-field). Note that a full storage cycle (1.5μsec) corresponds to two read-only storage cycles (750nsec).

The codes CM = 3, 4, or 5 specify read cycles. The addresses are supplied from the register pairs IJ, UV, and LT, respectively. A read cycle reads one byte of data from core storage into the storage data register R.

A write cycle regenerates the data from the storage data register R at the address supplied in the last read cycle.

A store cycle acts exactly as a write cycle except that it inhibits in the read cycle immediately preceding it the insertion of the data byte from storage into the R-register.

The CU-field specifies whether storage access should be to main storage (MS) or to a local storage of 256 bytes not explicitly addressable by the 360 language programmer.

Microprogram Sequencing and Branching. Each microprogram word is stored at a unique address in ROS. A 13-bit ROS address register (W3···W7, X0···X7) holds the address of the word being executed. For the symbolic representation of a microprogram (Figure 6) the ROS address is given in hexadecimal in the upper right corner, and the last two bits of this address are repeated in binary on the upper margin.

After execution of a microprogram step, the next sequential word will not be executed. Instead the address of the next word to be executed is derived as follows. The high five bits (W) remain the same, unless they are changed by a special command in the microword, not explained here (so-called module switching). The next six bits (X0···X5) are supplied from the CN-field (written in hexadecimal in the symbolic representation of Figure 6). The low two bits are set according to conditions specified in the CH and CL fields. X6 is set according to the condition specified by CH. For instance, if CH = 8, then the bit R2 is transferred to X6; if CH = 6, then X6 is set to one if in the last ALU operation a carry had occurred. It is set to zero if no carry had occurred. X7 is controlled by CL. If, for instance, CL = 0, then X7 is set to zero; if X7 = 5, then X7 is set to one if both digits in R are valid decimal digits (i.e., R0···R3 $\leq$ 9 and R4···R7 $\leq$ 9), X7 is set to zero if either digit in R is not a valid decimal digit (i.e., R0···R3 > 9 or R4···R7 > 9). This microprogram sequencing scheme allows a four-way branch after the execution of each microprogram word.

Status Bit Setting. The CS-field allows the unconditional or conditional setting of certain status bits to be specified, combined in Register S. If, for instance, CS = 3, then S4 is set to one if the result of the ALU operation performed in this microprogram cycle shows a zero in the high digit (i.e., Z0=Z1=Z2=Z3=0); S4 is set to zero otherwise. At the same time, S5 is set to one if the result of the ALU operation shows a zero in the low digit (i.e., Z4=Z5=Z6=Z7=0); S5 is set to zero otherwise. If CS = 9, then S2 is set to one if the result of the ALU operation is not zero (i.e., at least one of the bits Z0···Z7 is equal to 1). If the result of the ALU operation is zero, then S2 is not changed.

Constant Field. The 4-bit CK-field is used for various purposes. One instance explained in the ALU statement is to supply a constant B-source for an ALU operation. Other examples not explained here any further are the addressing of a few specific scratchpad local storage locations, module switching (replacement of the high part W of the ROS address), and the control of certain special functions.

Symbolic Representation of Microprograms. Microprograms are symbolically represented as a network of boxes (Figure 6) each representing a microword, connected by nets indicating the possible branching ways. Figure 7 gives an example of a microprogram (to be explained in the next section). There exist programming systems to aid in the development of microprograms. They contain symbolic translators to translate the contents of a box according to Figure 6 into the contents of the actual fields

Fig. 7. Microprogram for the operators **and**, **or**, and **then**.

of the microprogram word according to Figure 5. A drawing program generates documentation (Figure 7 is drawn with such a program). These systems usually also contain programs for simulation and generation of the actual ROS cards.

5. String Language Interpreter for EULER

The string language interpreter for EULER is entirely written in Model 30 microcode. It consists of a few microprogram steps to read the next sequential symbol from the program string and to do a function branch on the symbol and of a group of microprogram routines which perform the necessary operations for the program byte read. These routines also take care of dynamic type testing and stack pointer manipulations. The routines are equivalent to the routines described in the definition of the string language for EULER [1].

Figure 7 shows as an example, the microprogram to interpret the program string symbols **and** (internal representation X'52'[3]), **or** X'50' and **then** X'53'. These operators test if the highest entry in the stack is a value of type logical. The logical operators in EULER work in the FORTRAN sense, not in the ALGOL sense: if after the evaluation of the first operand the result is determined (**false** for **and,** **true** for **or**), then the second operand is not evaluated but skipped over. If an **and** operator finds the value **false,** then a branch occurs to the program address given in the two trailer bytes. If an **and** finds the value **true,** then it deletes this value from the stack and proceeds to the next symbol in the program string (to evaluate the second operand of **and**). Similarly if an **or** operator finds the value **true,** then a branch occurs to the program address given in the two trailer bytes. If an **or** finds the value **false,** then it deletes this value from the stack and proceeds to the next symbol in the program string. The **then** operator is a conditional branch code: it deletes the logical value from the stack. If this value was **false**, then a branch is taken to the program address given in the two trailer bytes. If this value was **true,** then the next symbol in the program string is executed.

The pointer to the symbol in the program string (the instruction counter) is located in the functionally associated pair of registers I and J in the Model 30. The pointer to the left-most byte of the highest entry in the stack (the stack pointer) is located in the two registers U and V in the Model 30.

In the following the individual steps in this microprogram are explained in more detail.

Address	*Location in Figure*	*Description*
1161:	C1:	The instruction counter IJ addresses main storage. The addressed byte in main storage is read out into the storage data register R. The instruction counter is updated by adding 1 to register J. A possible carry is saved to be added to 1.
1117:	C2:	The operator has been read out from main storage into R. It is also transferred (through the ALU) to register G. A four-way branch occurs on the two highest bits R0 and R1 of the operator. For the operators 52, 53, and 50 this branch goes to ROS word 1171, whereas other operators cause a branch to 1170, 1172, or 1173, indicated by the three lines not continued.
1171:	C3:	To complete the updating of the instruction counter, the carry from 1161 is added into I. The first byte of the highest entry of the stack is addressed by UV and read out into R. A further four-way branch on the operator is made (G2, G3). For our operators the branch goes to 115D.
115D:	C4:	The high order byte of the highest stack entry has been read out of storage into R. It contains the type of entry in the high digit and if this type was logical then it contains the value **true** (1) or **false** (0) in the second digit. This byte is tested by adding X'DO' to it and observing the result, ignoring the carry. S4 is set to 1 when the type was 3(logical) otherwise to 0. S5 is set to 1 when the low digit of this byte was 0 (value **false**), S5 is set to 0 when the low digit of this byte was 1 (value **true**). Another four-way branch occurs on the bits G4 and G5 of the operator. If the operator is 50(**or**), 51 (cannot occur), 52 (**and**), or 53(**then**), then a branch to 11C4 occurs.
11C4:	L4:	The next byte is read from the program string, it is the high byte of the two-byte program address trailing the operator. The instruction counter is updated again by adding a 1 to J, saving a possible carry. Another four-way branch occurs on the bit G6 of the operator and the value of the stack entry. If the operator was **and** or **then** (G6 = 1) and the value was **false** (S5 = 1), then branching to 11CB occurs; if the operator was **or** (G6 = 0) and the value was **true** (S5 = 0), then branching to 11C8 occurs. If the operator was **or** (G6 = 0) and the value was **false** (S5 = 1), then branching to 11C9 occurs. If the operator was **and** or **then** (G6 = 1) and the value was **true** (S5 = 0), then branching to 11CA occurs.
11CB:	G5:	This word is executed for the operators **and** and **then** when the value was **false.** Here the type test is made. If the type was not logical (S4 = 0), then a branch to 11C1 occurs. If the type was correct, then the microprogram proceeds to fetching the trailing program address (two bytes) to store it as the new instruction counter in IJ. This is done for the **and** operator (G7 = 0) in this word and the following two words 11C3 and 111E; for the **then** operator (G7 = 1) it is done in this word and the words 11C3 and 111F.

[3] X *'nn'* represents the hexadecimal number composed of the digits n ($n = 0,\ldots,9, A,\ldots,F$).

11C3, 111E:	J6, J7:	The two bytes trailing of the operators **and** or **or** are stored as the new instruction counter IJ. The operation is completed. The microprogram branches back to 1161 to read out the next operator.
11C3, 111F:	J6, L7:	The two bytes trailing of the operator **then** are stored as the new instruction counter in IJ. The carry-saving bit S3 is forced to zero.
11CE, 1144:	N8, N9:	The stackpointer is decremented by four (the operator '-' means complement add) which in effect deletes the highest entry from the stack. Observe that when these two words are entered from 111F (**then** operator with value **false**) the microprogram will not go through 1145 because we have forced S3 to zero in 111F. The operation is completed, and the microprogram branches back to 1161 to read out the next operator.
11C8:	J5:	This word is executed for the operator **or** when the value was **true**. Similarly as in 11CB, the typetest is taken. For types not logical a branch to 11C1 occurs. If the type was correct, then the microprogram proceeds to fetching the trailing program address (two bytes) to store it as the new instruction counter in IJ (words 11C3, 111E).
11C9:	N5:	This word is executed for the operator **or** when the value was **false.** A typetest is made. If the type was correct, then the trailing program address is skipped and IJ is updated by 1 twice in 11C4, 11C9 (possible carries out of J handled in 11CF or 1145). The stackpointer is decremented by four in 11CE, 1144.
11CA:	Q5:	This word is executed for the operators **and** and **then** when the value was **true**. A typetest is made. If the type was correct then the trailing address is skipped, IJ is updated by 1 twice in 11C4, 11CA (possible carries out of J handled in 11CF or 1145). The stackpointer is decremented by four in 11CE, 1144.
11C1, 11CC, 11CD:	G6,L6,N6	These words are executed when a typetest occurs. An error code 01 is set up in L and a branch occurs to the error routine not drawn here.

It can be seen from Figure 7 that the execution times of the microprograms including the readout of the operator (I-Cycle) are the following:

and	6μsec[4] (8 microprogram steps)
or	6μsec (8 microprogram steps)
then	6μsec for value **true** (8 microprogram steps)
	7.5μsec for value **false** (10 microprogram steps)

[4] The cases where carries occur in the IJ and UV updating are disregarded for timing purposes.

In order to compare this with a hypothetical EULER system for System/360 language, let us assume that the compiler produces in-line code (which probably will give the highest performance although it will be very wasteful with respect to storage space). Then a reasonable sequence for **and** might be:

```
CLI  0 (STACK), LOGFALSE
BE   ANDFALSE
CLI  0 (STACK), LOGTRUE
BNE  TYPEERR
SH   STACK, = '4'
```

Timing: **true**: 90μsec; **false**: 32μsec.

This comparison seems to indicate that the microprogram interpreter is about an order of magnitude faster than the equivalent program in 360 language. However, this comparison will only yield such a high factor for functions of EULER which do not have simple System/360 language counterparts (as for instance the list-operators, begin-, end-, and procedure-call-operator) or where the overhead for dynamic testing and stackpointer manipulation is heavy as in the above example of the logical operations. For functions which do have System/360 language counterparts and which are slower so that the overhead is relatively lighter as, for instance, arithmetic operations (especially for real numbers), the microprogrammed interpreter will still be faster than the System/360 language program, but not by a factor of 10.

The total ROS space requirement for the String Language Interpreter is:

Coded routines	1000 microwords
Routines for real number handling	500 microwords (estimated)
Divide, Exponentiation, etc.	400 microwords (estimated)
Garbage collector	600 microwords (estimated)
	2500 microwords

6. EULER Compiler

The translator to translate EULER source language into the Reverse Polish String Language is a one-pass, syntax-driven compiler. The syntax of the language and the precedence functions F and G over the terminal and nonterminal symbols are stored in table form is Model 30 main storage. There is also main storage space reserved for translation tables for character delimiters and word delimiters and for a compile time stack, a name table, and, of course, for the compiled code. All these areas are at fixed storage locations because of the experimental nature of the system.

The microprogram consists of the following parts:

1. A routine reads the next input character from the input buffer to translate it to a 1-byte internal format, if it is a delimiter, or to collect it into a name buffer if it is part of an identifier, or to convert it to hexadecimal if it is

part of a numeric constant and to collect the number into a buffer. This "prescan" requires 100+ microwords.

2. As soon as an input unit is collected (delimiter, identifier, number) the main parsing loop is entered which makes use of the precedence tables and the syntax table in main storage. This syntactic analyzer loop requires 100− microwords.

3. When the parsing loop identifies a syntactic unit to be reduced, it calls the appropriate generation routine which performs essentially the functions described as the semantic interpretation rules in the EULER definition. The microprogram space required for these programs amounts to approximately 250 ROS words.

4. If a syntactic error is detected, the system signals an error and does not try to continue with the compilation process. Though this procedure is totally inadequate for a practically useful system, it was deemed sufficient to prove the essential point. For this minimum error analysis and for linkage to the 360 microprograms (IOCP), approximately 60 microwords are required.

The total compiler microprogram space is therefore approximately 500 ROS words. The total main storage space required is approximately 1200 bytes.

The speed of this compiler is limited by the speed of the card-reader of the system (1000 cards/minute). This excellent performance has three main reasons: (1) EULER as a simple precedence language is a language extremely easy to compile. (2) The functions of a compiler are mainly of a table lookup and bit and byte-testing type. Microprogramming is extremely well-suited for these kinds of operations. (3) Since the target language is String Code and not, for example, 360 Machine Language, the generative part of the compiler is relatively short.

It is very difficult to assess the individual contributions of these three main reasons to the high compiler performance. Therefore, it is not possible at this stage to make a statement as to whether the nature of the language EULER or the fact that the compiler is microprogrammed is the dominant factor.

7. Development of the Microprogram

Since there is no higher level language to express microprogram procedures and no compiler to compile microcode, the microprograms were written in the symbolic language explained in Figure 6. Actually the process was a hand translation of the algorithms in the EULER definition to the symbolic microprogram language. The microprograms were translated into actual microcode and simulated before they were put on the System/360 Model 30 by means of a general microprogram development system.

8. Outlook and General Discussion

It is hoped that the development of this experimental system for EULER shows that with the help of microprogramming we can create systems for higher level languages or special applications, which utilize existing computer hardware to a much higher degree than conventional programming systems.

Among the thoughts which are raised by this scheme are the following:

1. There should be an investigation to determine the ideal directly interpretable languages which correspond to higher level languages. Although several attempts have been made to define string languages for interpretive systems (for instance in [1, 4]), to the author's knowledge no work has been published which attacks this question in a general and theoretically founded manner.

2. A proliferation of interpretive languages and the development of microprogrammed interpreters can be justified when better tools are developed to reduce the cost of microprogramming. It is necessary that we be able to express microprogramming concepts (and also machine design concepts) in a higher level language form and that we develop compilers which translate the microprograms from higher level language form to actual microcode. Also, good microprogram simulation and debugging tools are called for.

3. The whole relationship between programming, microprogramming, and machine design should be viewed with a common denominator: how should the tradeoffs be made such that the ultimate goal can be reached more effectively, ... how to solve a user's problem? Green [5] offers some thinking in this direction but the state of the art has to progress further before we will have a complete understanding of what these relationships and tradeoffs are.

Acknowledgment. I wish to thank Jack Carman, who wrote the I/O Control Program and the Operating System linkage for the EULER system and Miss Sheila Morrison who helped prepare the figures. I am also grateful for the valuable criticism offered by the referee, W. C. McGee, as well as by Professor N. Wirth and E. Satterthwaite.

RECEIVED DECEMBER, 1966; REVISED MAY, 1967

REFERENCES

1. WIRTH, N., AND WEBER, H. EULER: A generalization of ALGOL, and its formal definition: Pt. I, *Comm ACM 9*, 1 (Jan. 1966), 13–25; Pt. II, *Comm ACM 9*, 2 (Feb. 1966), 89–99.
2. FAGG, P., BROWN, J. L., HIPP, J. A., DOODY, D. T., FAIRCLOUGH, J. W., AND GREENE, J. IBM System/360 engineering. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 28, pp. 205–231.
3. HAINES, L. H. Serial compilation and the 1401 FORTRAN compiler. *IBM Sys. J. 4* 1 (Jan. 1965), 73–80. See also: FORTRAN specifications and operating procedures, IBM 1401 IBM Systems Ref. Lib. C24-1455-2.
4. MELBOURNE, A. J., AND PUGMIRE, J. M. A small computer for the direct processing of FORTRAN statements. *Comput. J. 8*(April 1965), 24–27.
5. GREEN, J. Microprogramming, emulators and programming languages. *Comm ACM 9*, 3 (Mar. 1966), 230–231.