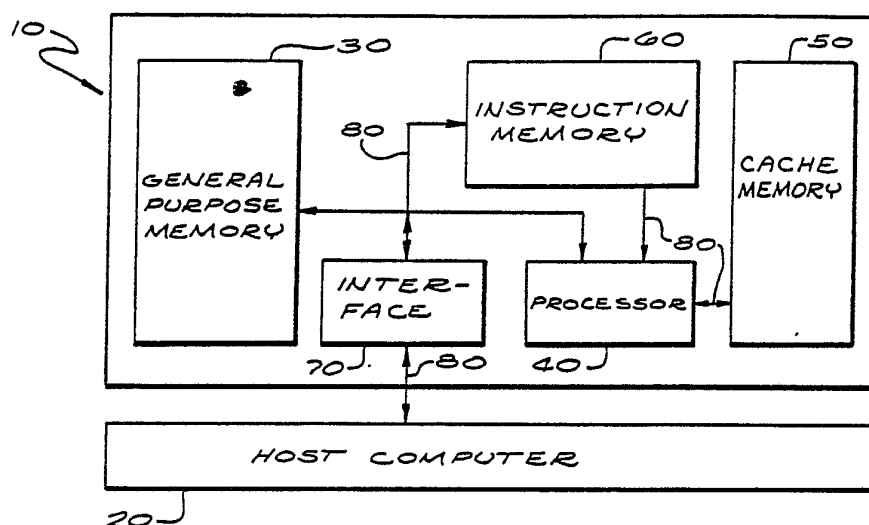




INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁴ : G06F 9/34, 9/44	A1	(11) International Publication Number: WO 88/ 05190 (43) International Publication Date: 14 July 1988 (14.07.88)
(21) International Application Number: PCT/US87/03444 (22) International Filing Date: 28 December 1987 (28.12.87) (31) Priority Application Number: 000,714 (32) Priority Date: 6 January 1987 (06.01.87) (33) Priority Country: US (71) Applicant: INTERNATIONAL META SYSTEMS, INC. [US/US]; 9841 Airport Boulevard, Suite 1508, Los Angeles, CA 90045 (US). (72) Inventors: SPECKHARD, Arthur, E. ; 28205 S. Ridegefern Ct., Rancho Palos Verdes, CA 90274 (US). THAMES, Joseph, M. ; 5406 Via Del Valle, Torrance, CA 90505 (US).		(74) Agent: CARSON, M., John; 201 North Figueroa Street, 5th Floor, Los Angeles, CA 90012 (US). (81) Designated States: DE (European patent), FR (European patent), GB (European patent), IT (European patent), JP. Published <i>With international search report.</i> <i>With amended claims.</i>

(54) Title: MICROPROGRAMMABLE LANGUAGE EMULATION SYSTEM**(57) Abstract**

A data processing system executes a high level computer language program by encoding statements of the program into variable length tokens and then executing the tokens. Each token is a variable length bit field having a value representative of a semantic element of a program statement and a length representative of the context of the semantic element. A single very large scale integrated circuit (VLSI) processor (40) is microprogrammed to execute the encoded program in a pipeline manner and may be used in conjunction with a host computer (20), such as an IBM PC AT or equivalent system.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	FR	France	ML	Mali
AU	Australia	GA	Gabon	MR	Mauritania
BB	Barbados	GB	United Kingdom	MW	Malawi
BE	Belgium	HU	Hungary	NL	Netherlands
BG	Bulgaria	IT	Italy	NO	Norway
BJ	Benin	JP	Japan	RO	Romania
BR	Brazil	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic	KR	Republic of Korea	SE	Sweden
CG	Congo	LI	Liechtenstein	SN	Senegal
CH	Switzerland	LK	Sri Lanka	SU	Soviet Union
CM	Cameroon	LU	Luxembourg	TD	Chad
DE	Germany, Federal Republic of	MC	Monaco	TG	Togo
DK	Denmark	MG	Madagascar	US	United States of America
FI	Finland				

MICROPROGRAMMABLE LANGUAGE EMULATION SYSTEMBACKGROUND OF THE INVENTION

5 The present invention generally relates to a system for execution of high level computer language programs. More particularly, it relates to a system for emulating high level computer languages and executing programs written in such languages.

10 Computer languages have traditionally been divided into two classes: programming languages and machine languages. As the name implies, machine languages have been used to refer to machine elements of computers and have been directed to each action to be performed by the computers for
15 which they are written. Programming languages, on the other hand, are generally considered to be "high level" languages because their operators and constructs correspond more to properties of applications or problems to be solved rather than actual machine functions or physical elements.

20 Because high level languages typically do not make reference to actual machine hardware, high level languages have been translated into machine languages in order for computers to execute the statements of high level languages. The traditional method for translating high level languages
25 has been to compile several machine language instructions for each high level language statement. Multiple machine language instructions have been used because the logic of high level languages typically cannot be expressed in one-to-one correspondence with machine language instructions.

30 A disadvantage of compiling several machine language instructions for each high level language statement is that most computers execute only one instruction at a time. Moreover, each instruction must be fetched from a memory location one at a time, presenting a "bottleneck"

between memory access time and machine processing speed. Recent improvements in very large scale integrated (VLSI) circuit technology have dramatically increased processing speed and illuminated the "bottleneck" problem.

5

MICROFICHE APPENDICES

Microfiche appendices, which constitute a part of this specification, are as follows:

10

MICROFICHE APPENDIX A is a computer program listing of the encoder program of the preferred embodiment, contained on three microfiche having 201 frames; and

15

MICROFICHE APPENDIX B is a computer program listing of the emulator program of the preferred embodiment, contained on 2 microfiche having 162 frames.

SUMMARY OF THE INVENTION

20

The present invention solves the problems associated with the compiling of high level languages by encoding such languages in variable length tokens representative of characteristics intrinsic to such languages. The tokens are then executed by a processor which is microprogrammed to emulate a high level language.

25

The system of the present invention may be used independently or in conjunction with a host computer system, such as an IBM PC AT or equivalent system, to provide high speed processing of application programs written in a multiplicity of high level languages. Because the processor of the present invention is microprogrammable, encoder and emulator programs written for specific languages may be used as microcode for the processor.

30

It is to be understood that the following description of the preferred embodiment is illustrative of the present invention but other embodiments are possible

without departing from the spirit and scope of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

5

The drawings, which constitute a part of this specification, are as follows:

FIG. 1 is a block diagram of apparatus of the preferred embodiment;

10

FIG. 2 is a block diagram of a program procedure;

FIG. 3 is a logic flow-chart showing the word padding function of the preferred embodiment;

15

FIG. 4 is a sample listing of tokens of the preferred embodiment, which are used to represent a high level language statement;

FIG. 5 is a conceptualized depiction of the branching operation of the preferred embodiment;

20

FIGs. 6a and 6b are graphic representations of alternative instruction formats of the processor of the preferred embodiment; and

FIG. 7 is a block diagram of the processor of the preferred embodiment;

DETAILED DESCRIPTION OF THE DRAWINGS

25

Referring initially to FIG. 1, a general overview of the preferred embodiment will now be described. An expansion board 10 is added to a host computer 20, such as an IBM PC AT or equivalent system. The expansion board 10 includes a main memory 30, a processor 40, a cache memory 50, an instruction memory 60, an interface 70 with the host computer 20 and bus and control lines 80.

30

Electrical signals representative of a high level program are loaded into the main memory 30 by the processor 40 from the host computer 20 through the interface 70 and bus

lines 80. The main memory 30 is a dynamic random access memory unit having a storage capacity of approximately 1 megabyte and information is stored in the general purpose memory 30 in 32-bit words.

5 An encoder program is loaded into the instruction memory 60 by the host computer 20 through the interface 70 and bus lines 80. The instruction memory is a loadable read only memory (ROM) unit having a storage capacity of 64K 32-bit words. A listing of a representative encoder program is
10 attached to this disclosure as Microfiche Appendix A and incorporated by reference. The encoder program causes the processor 40 to fetch statements of the high level language program from the main memory 30, encode each statement in a representative stream of variable length bit fields
15 ("tokens") without regard to word boundaries, and store the encoded statements in the main memory 30.

 After the high level language statements have been encoded and stored, an emulator program is loaded into the instruction memory 60 by the host computer 20 through the
20 interface 70 and bus lines 80. A listing of a representative emulator program is attached to this disclosure as Microfiche Appendix B and incorporated by reference. The emulator program causes the microprocessor to fetch the encoded statements from the main memory 30 and interprets the tokens
25 of the encoded statements into microcode instructions resulting in execution of the high level language. That is, the instruction tokens executed by the emulator program have a syntax and data structure resembling the syntax and data structure of the high level language rather than the physical
30 elements of the processor. Instruction tokens are packed into the main memory 30 and subsequently fetched and executed by the processor 40.

 Referring now to FIG. 2, the instruction token architecture for encoding a high level language will be described. For purposes of illustration, FORTRAN 77 is used

as a representative high level language. A program written in that language is generally a set of programming units known as "procedures." Each procedure 100 has a specific function to perform and typically comprises three major parts: a header 110; a code body 120; and a data contour 130. The header 110 contains a table which describes the bit size and storage locations of the code body 120 and the data contour 130. The code body 120 contains the logic statements of the procedure and the data contour 130 contains data or storage locations of data to be used in the procedure.

In the preferred embodiment, absolute addresses are not used to provide location references to elements of the code body 120 or data contour 130. Rather, all location references are relative to fixed positions in the data contour 130 or the header 110. Thus, machine references are not directly made. An exception exists, however, for global references which are used to refer to other procedures and to common or global variables.

Each statement of the high level language is represented in a stream of instruction tokens generally without regard to word size, again allowing for independence from machine constraints. Each token is a variable length bit field representable by an integer pair having the following general form. The first integer of the pair is a constant of the emulator program, fixed by the order of tokens in the language representation, which indicates the length of the token in bits. The second integer is the value of the token, representative of an operator or operand corresponding to characteristics that are intrinsic to the execution of the high level language. In combination, the integer pair provides a switching context within the emulator program potentially having a branch for each integer in the set of all integers representable by the token of the specified length. That is, during execution, the emulator program interprets each token to cause the processor 40 to

branch to locations in analagous fashion to a FORTRAN computed GO TO statement or a PASCAL CASE statement.

For convenience, the token stream format is presented herein in a left-to-right ordering but in actual implementation, the ordering is right-to-left. That is, token streams fill memory words of the main memory 30 beginning at the least significant bit (bit position 0) of a memory word and ending at the most significant bit (bit position 31). Although the tokens are generated without regard to word size, they must be packed in memory according to the word boundaries of the main memory 30, that is, 32 bits. Accordingly, as shown in FIG. 3, the number of bits remaining in a memory word is calculated (200) for each token stored. If a token requires more than the number of bits remaining in a word (210), that word is padded with 0's (220), which are ignored by the emulator program, and the token is packed at the beginning of the next word (230). Such padding allows the generation of tokens to be free from considerations of word size.

The primary operators are shown in TABLE I below:

TABLE I

	<u>Primary Operators</u>	<u>Statement</u>
25	{6:0}	invalid
	{6:1}	Assignment
	{6:3}	ASSIGN
	{6:4}	BACKSPACE
	{6:5}	CALL
	{6:6}	CLOSE
	{6:7}	DECODE
30	{6:8}	DELETE
	{6:9}	DO (For)
	{6:10}	DO WHILE
	{6:11}	ENCODE
	{6:12}	END DO
	{6:13}	ENDFILE
	{6:14}	FIND
	{6:15}	FORMAT
	{6:16}	GO TO

	{6:17}	Computed GO TO
	{6:18}	Assigned GO TO
	{6:19}	Arithmetic IF
	{6:20}	Logical IF
	{6:21}	ELSE
	{6:22}	END IF
5	{6:23}	INQUIRE
	{6:24}	OPEN
	{6:25}	OPTIONS
	{6:26}	PAUSE
	{6:27}	READ
	{6:28}	RETURN
	{6:29}	REWIND
10	{6:30}	REWRITE
	{6:31}	SAVE
	{6:32}	Statement Function
	{6:33}	STOP
	{6:34}	UNLOCK
	{6:35}	WRITE
	{6:36}	DATA
	{6:37}	ENTRY
15	{6:38}	FUNCTION
	{6:39}	SUBROUTINE
	{6:40}	PROGRAM

The first integer in each primary operator token representation indicates that the instruction token is six bits in length. This integer is not part of the instruction token stream, but is part of the emulator program which executes the token stream. Its value is predictable according to the ordering of tokens in the language representation. The second integer (the token value) indicates which statement of the high level language is being encoded. For each of these operators there are suboperations corresponding to the suboperations of the high level language, thus providing isomorphic representation of the high level language. A common suboperation is the "expression," which contains a blend of operators and operands. An operand may in turn correspond to syntax names or literals as in the high level language.

Expressions are initiated by preceding operations which imply the beginning of an expression and concluded with

an "expression-end" operator. All operators, variable-
operands and literal operands within the expression are
prefaced with a discriminator token as shown in TABLE II
below:

5

TABLE II

	<u>Discriminator</u>	<u>Meaning</u>
	{2:0}	Invalid
10	{2:1}	Operator
	{2:2}	Variable Operand
	{2:3}	Literal Operand

Secondary operators, used in expressions, are shown in TABLE
III below, together with an operator discriminator token:

15

TABLE III

	<u>Secondary Operators</u>	<u>Use</u>
	{2:1}{5:0}	Invalid
20	{2:1}{5:1}	Generic addition
	{2:1}{5:2}	Generic subtraction
	{2:1}{5:3}	Generic multiplication
	{2:1}{5:4}	Generic division
	{2:1}{5:5}	Logical AND
	{2:1}{5:6}	Logical inclusive OR
	{2:1}{5:7}	Logical exclusive OR
25	{2:1}{5:8}	Logical negation
	{2:1}{5:9}	Equal
	{2:1}{5:10}	Not equal
	{2:1}{5:11}	Greater than
	{2:1}{5:12}	Greater than or equal
	{2:1}{5:13}	Less than
	{2:1}{5:14}	Less than or equal
30	{2:1}{5:15}	Function call
	{2:1}{5:16}	Through (RHS)
	{2:1}{5:17}	Index (RHS)
	{2:1}{5:18}	Through (LHS)
	{2:1}{5:19}	Index (LHS)
	{2:1}{5:20}	End of list
	{2:1}{5:21}	Expression end
	{2:1}{5:22}	Integer remainder
	{2:1}{5:23}	Real square root
	{2:1}{5:24}	Real sine

{2:1}{5:25}	Real cosine
{2:1}{5:26}	Generic absolute value
{2:1}{5:27}	Generic exponentiation

5

Operands (data references) are token structures corresponding to syntax names that are used to reference the content of data structures. However, operand references are not simply memory addresses. Rather, the mapping of references to data content is a dynamic process. Two kinds of operand references are employed in code-body 120 expressions: variable references and name references. Variable references are references to variable data structures while name references correspond to names of variable data structures. In general, variable references 15 are used to retrieve data, whereas name references are used to store data. A name is considered a literal.

20

The reference structure of these two types of operands is as follows:

```
Variable: {2:2}{4:class}{primary}[{secondary}]
Name:     {2:3}{4:3}{4:class}{primary}[{secondary}]
```

25

The second field of the name reference is actually a type code {4:3} referring to type "name," since a name is a literal data structure. The reference structure has a four-bit class field, a variable format primary reference and may 30 have a secondary reference relative to the primary one (common variables only). The primary reference is an index to one of the tables in the contour 130 portion of the object program, as designated by the class field. The four-bit class field and associated class designations are listed in TABLE IV below:

TABLE IV

<u>Class Code</u>	<u>Class Designation</u>
{4:0=inv}	Invalid
5 {4:1=LCV}	Local Variable
{4:2=CMV}	Common Variable
{4:3=PAR}	Parameter Variable
{4:4=ITV}	Intrinsic Variable
{4:5=LCL}	Local Label
{4:6=GLL}	Global Label
{4:7=ITL}	Intrinsic Label
10 {4:8=NLG}	Namelist Group Name

The primary reference format has a two bit length code and a value subfield. The length of the value subfield is encoded in the length specification shown in TABLE V below:

TABLE V

<u>Length Code</u>	<u>Value Encoding</u>
20 {2:0}	{3:value}
{2:1}	{5:value}
{2:2}	{7:value}
{2:3}	{12:value}

25 The preceding class code and the length code are contiguous in the same word so that a zero length code may be distinguished from a zero pad, mentioned above.

In all reference classes except common variables, only a primary reference exists. In these cases, references have the token structure shown in TABLE VI below:

TABLE VI

Variable:	{2:2}{4:class}{length-code}{value}
Name:	{2:3}{4:3}{4:class}{length-code}{value}

The value represents a relative pointer into one of the tables in the contour 130 as specified by the class field.

Common variable references have the token structure shown in TABLE VII below:

5

TABLE VII

	Variable: {2:2}{4:class}{length-code}{block-value} {offset-value}
10	Name: {2:3}{4:3}{4:class}{length-code} {block-value}{offset-value}

The length code applies only to the block-value. The offset value is a "word encoded" literal, meaning that its size is determined by the number of remaining bits in the word containing the preceding fields. If enough bits remain to contain the value (which cannot be zero), these bits are used as the offset-value field. Otherwise, the remaining bits are zero, and the next full word is used as the offset field.

Literal operands and operand data structures have similar formats. Literal operands exist in the code body 120. Operand data structures, if they exist prior to execution, are stored in the data contour 130. Otherwise, they are created and stored dynamically and are referenced indirectly through the data contour 130 tables. Data structures are prefaced by a four-bit type code, as shown in TABLE VIII below:

30

TABLE VIII

<u>LITERALS</u>		<u>IN CONTOUR</u>	
{4:0}	Invalid	{4:1}	Logical
{4:1}	Logical	{4:2}	Byte Character
{4:2}	Character string	{4:3}	Name
{4:3}	Name	{4:4}	Integer*2
{4:5}	Integer	{4:5}	Integer*4

{4:6}	Real*4	{4:6}	Real*4
{4:7}	Real*8	{4:7}	Real*8
{4:8}	Real*16	{4:8}	Real*16
{4:9}	Complex*8	{4:9}	Complex*16
{4:10}	Complex*16	{4:10}	Complex*16
		{4:11}	Array

5

Such data structures possess great variability in length and may not fit in a single word of memory. If not, the entire value, including the sign bit is stored in the next memory word. The low order bit of the data structure represents the sign of the value, containing a 0 for a positive sign or a 1 for a negative sign. The literal value of zero has a negative sign bit (1) to distinguish it from a 0 pad field, which is ignored by the emulator program. Thus, {000001} represents a literal value of zero, while {000000} represents a pad field.

Referring now to FIG. 4, an example of an encoded high level language statement will be described. The statement 300 is a FORTRAN assignment statement assigning the sum of 10 + 5 to the variable A. The first token 310 is a primary operator having a length of six bits and indicating an assignment statement, as shown in TABLE I above. The second token 320 is a literal operand discriminator, as shown in TABLE II above, which initiates an expression. The third token 330 indicates that the literal operand is an integer, as shown in TABLE VIII above. The fourth token 340 has a length of 20 bits to fill the remainder of a 32 bit memory word and a value of 20 which indicates that its true value of 10 has been shifted to add a positive sign bit to the zero bit position of the memory word. Such single position shifting has the appearance of multiplying the value by 2 since it is encoded in binary form.

The fifth token 350 is another literal operand discriminator to preface a literal value. The sixth token 360 indicates that the literal operand is an integer and the

seventh token 370 has a length of 26 bits to fill the remainder of a 32-bit memory word. The literal value of the seventh token 370 is 10 which indicates that a positive sign bit was added to the memory word to shift the true value of 5
5 by 1 bit.

The eighth token 380 is an operator discriminator to indicate that an operation is to be performed in the expression. It follows the literal operand token groups described above to indicate Polish post-fix notation. That
10 is, the operator is applied to operands which precede it. During execution of the encoded statement, the literal operands are loaded into a push-down stack so that operators are applied to them on a last-in-first-out basis. The ninth token 390 indicates an addition operation is to be performed,
15 as shown in TABLE III above.

The tenth token 400 is another operator discriminator and the eleventh token 410 indicates an end to the expression, as shown in TABLE III above. The twelfth token 420 initiates a new expression containing a literal
20 operand as shown in TABLE II. The thirteenth token 430 indicates that the literal operand is a name reference as shown in TABLE VIII. (The structure of this reference is shown in TABLE VI, in which its first two tokens correspond to 420 and 430 in FIG. 4.)

25 The fourteenth token 440 is the class code of the name reference, indicating the name of a local variable as shown in TABLE IV. The fifteenth token 450 is the length code of the reference as shown in TABLE V, indicating that the subsequent (sixteenth) value token 460 is 5 bits in
30 length. The value token 460 is a relative pointer to the location of the local variable in the data contour 130, as shown in FIG. 2. Its value, one, indicates the location of the first local variable (local variable A 130G1) in the Local Variables Table 130G of the data contour 130 located by the local variables offset pointer in the Header 110. The

seventeenth token 470 is another operator discriminator and the eighteenth token 480 is an Expression-End operator, as shown in TABLE III.

5 Recall that the integer pair symbolizing each token represents a logical switching context in the emulator program containing a branch corresponding to each integer in the set of integers representable by the token of the specified length. As shown in FIG. 5, the branching operation of the emulator program may be conceptualized as a
10 logical tree system. Considering the example assignment statement referred to in FIG. 4, the token stream defines a path through a hierarchy of integer pairs, which achieves the execution of the statement.

15 The primary operator token, (6:1), is a member of the highest order set which represents the statement context of the high-level language. The integer one (500), indicated by the value of the token, designates a branch to the Assign. (primary) operator, as shown in TABLE I. The Assign operator automatically invokes expression subcontext which begins with
20 discriminator subcontext, as shown in TABLE II.

The length of the discriminator token (510) allows three branches (zero values are invalid in most contexts since zeros are used as padding, causing the emulator program to proceed to the next word). The value of the discriminator
25 token (510), three, selects the third branch indicating that a literal data structure follows.

The literal data structure begins with a type subcontext as shown in TABLE VIII. The length of the type token (520), four bits, allows up to 15 type branches
30 (excluding the zero value). The value, four, of the type token (520) selects the integer type. The literal data structure is concluded with the value of the integer. If enough bits remain in the word containing the preceding tokens to represent the integer value, then the remainder of the word is used. Otherwise, the remainder of the word is

zero-filled (and ignored by the emulator program) and the subsequent full word is used (as a token) to contain the value of the integer. In the example (530), twenty bits remain in the 32-bit word, and are used as the value token.

5 The value itself consists of a zero in the low order bit (the sign bit) indicating a plus sign, and the value 10 in the high-order nineteen bits.

A hierarchy of switching contexts exist in the emulator for each primary operator of the high level
10 language. The characteristics of the high level language are thus directly represented by the content and ordering of token streams. Branches are made isomorphically to the high level language so it is unnecessary to compile multiple machine instructions to interpret the logic of high level
15 language statements. Rather, such statements are executed in direct correspondence with their intrinsic characteristics as represented by the token streams.

An emulator program, such as shown in Microfiche Appendix B and incorporated herein, is loaded into the
20 instruction memory 50 and causes the processor 40 to execute token streams. The emulator program is written specifically to support the language of the program being executed. That is, each high level language requires its own emulator program in order to provide direct interpretation of tokens
25 via micro-programmed branching.

The emulator program interprets token streams using microcode instructions of the machine language of the processor 40. In the preferred embodiment, the processor 40 has an instruction set of 24 hardware operations that may be
30 combined into a composite (dual) instruction format having a left hand side (LHS) and a right hand side (RHS), as shown in FIG. 6a. The composite instruction is 32 bits wide and contains seven fields. Alternatively, the instruction may have only a LHS and contain only five fields, as shown in FIG. 6b. A composite instruction is designated by F=0 in the

F field at bit position 20 and a LHS-only instruction is designated by F=1 in the F field.

The LHS portion of an instruction contains an arithmetic, logic or shift operation between two operands with the result assigned to a third operand. The RHS portion of an instruction contains a second operation, including external bus instructions, subroutine link/return skips, transfers, or memory indexing.

A three address instruction format is used for LHS operations, having the following symbolic form:

$$T := A \text{ op } B$$

wherein a register specified by the "T" field is assigned (:=) the value of some binary operation (op) performed on the contents of the register specified by the "A" field and the contents specified by the "B" field. Considering the example FORTRAN assignment statement of FIG. 4, $A = 10 + 5$, the T field of an LHS instruction specifies a register containing the local variable "A," the A field specifies a register containing the value of 10 and the B field specifies a register containing the value of 5.

The 24 operations implemented in the processor are designated by the Operator (Op) and C fields, as shown TABLE IX below:

TABLE IX

<u>Operator Field</u>		
Op=0,1	Left Shift A by B	(0: rotational, 1: logical)
Op=2,3	Addition, A+B	(2: B=short, 3: B=long)
Op=4,5	Subtraction, A-B	(4: B=short, 5: B=long)
Op=6,7	Reverse Subtraction, B-A	(6: B=short, 7: B=long)
Op=8,9	Exclusive OR, A <u>xor</u> B	(8: B=short,

	Op=A,B	Logical OR, A <u>or</u> B	9: B=long) (A: B=short, B: B=long)
	Op=C,D	Logical AND, A <u>and</u> B	(C: B=short, D: B=long)
5	Op=E,F	Right Shift A by B	(E: rotational, F: logical)

C Field

	C=0	Skip conditionally
	C=1	Load K Register
	C=2	External Bus Operation Initiation - From/To
10	C=3	External Bus Operation Initiation - Emit/Receive
	C=4	Link to Subroutine
	C=5	Link Conditional To Subroutine
	C=6	Return From Subroutine
	C=7	Transfer Unconditionally
	C=8	Conditional Transfer on Overflow Bit = 0
	C=9	Conditional Transfer on Overflow Bit = 1
15	C=A	Conditional Transfer on Bit 0 = 0
	C=B	Conditional Transfer on Bit 0 = 1
	C=C	Conditional Transfer on Zero
	C=D	Conditional Transfer on Non-Zero
	C=E	Conditional Transfer on Bit 31 = 0
	C=F	Conditional Transfer on Bit 31 = 1

20 The implementation of certain of the operations listed above will be described in connection with a description of the processor 40 of the preferred embodiment below. However, it is to be recognized that the operations listed above are generally understood by those of ordinary skill in the art.

25 Referring now to FIG. 7, the processor 40 of the preferred embodiment is contained on a single very large scale integrated circuit (VLSI) silicon chip 800. It executes instructions in a "pipeline" manner in four phases Ph I, Ph II, Ph III and Ph IV. That is, four sequential

30 instructions are concurrently executed in one of the four phases, each phase being one clock cycle in duration. As execution of an instruction is completed, it exits the pipeline and a new instruction enters it. The intermediate instructions simultaneously advance to their next phase of execution.

During the first phase Ph I, an instruction is fetched from the instruction memory 50 and loaded into an instruction register 810. The address of the instruction is indicated by a location counter 830 which is either
5 incremented sequentially or given values by transfer, return or conditional transfer instructions.

The instruction is then decoded in the second phase Ph II by an instruction decoder 820. The LHS is decoded for the Operator, A and B fields but not for the T field, which
10 is passed unaltered to the next phase. The A and B fields designate registers in a general register file 840, or literal registers 850 and 860, whose values are passed to an A Register 870 and a B Register 880. The Operator field is passed to a Opcode Decoder 890 which determines which
15 operation is specified by the Operator field.

The RHS is also decoded during the second phase Ph II, but only the Unconditional Transfer, Link, Link Conditional, Return and Load K Register instructions are acted upon during the second phase. All other RHS
20 instructions are passed to the next phase. For the Unconditional Transfer instruction, the address field of the instruction is used to select the next value for the location counter 830, which references the address for the instruction to be fetched immediately after the instruction in Ph I
25 advances. The Link and Link Conditional instructions are executed in similar fashion to the Unconditional Transfer instruction but in addition they "push" the accompanying location counter value onto the Link Stack Register 900 for subsequent use by the Return instruction. That instruction
30 "pops" a value from the Link Stack Register 900 and adds the value of its address field to the popped value. The sum is used as the next value of the location counter 830. The Load K Register instruction loads its address field into the K Register 910, which is a special purpose register used in

conjunction with a Memory Address Register 920 to reference the cache memory 60.

During the third phase Ph III, only the LHS instruction is acted upon. The values selected in the second phase Ph II are operated upon as specified by the Operator field and the result is sent to an X Register 930, which gives the succeeding instruction access to that result when it advances in the next clock cycle. To illustrate, the sum $R1 + R2 + R3$ may be assigned to R5 by using the instruction $R4 := R3 + R2$ followed by $R5 := X + R1$. The result of the $R3 + R2$ operation is stored in the X Register 930 and then added to R1 when the follow up instruction advances into the third phase Ph III.

During the fourth phase Ph IV, the result of the LHS instruction held in the X Register 930 is stored in a general register or special register as specified by the T field. It is also used for conditional transfer or skip testing by the Test and Skip Logic 940. If a transfer or skip is indicated by result, the address field of the instruction is used to select the next value of the location counter 830 and causes the instructions in the first three phases to be inhibited.

The processor 40 communicates with the cache memory 60 through a Cache Memory Interface 950. It communicates with external systems through External Bus Logic 960 which is connected to a 32-bit-wide bi-directional bus 970. The interface 70 between the processor 40 and the host computer 20 is connected to the external bus 970 and responds to commands under control of the program in the instruction memory 50. Although the external bus 970 is 32-bits wide, only bits 15-0 and parity bits 1-0 are used for communicating with the interface 70. Two read and two write commands are implemented as follows:

TABLE X

Inputs From The External Bus

<u>Address</u>	<u>Function</u>
SS +0	Write Data
SS +1	Write Control

Outputs To The External Bus

<u>Address</u>	<u>Function</u>
SS +0	Read Data Register
SS +1	Read Status

The letters "ss" refer to the subsystem address of the interface 70. The Write Data function causes 16 bits of data to be sent from the processor 40 to the interface 70. A "data" bit in a status register of the interface 70 is also set to indicate that data has been transmitted. However, if either the data bit or control is already set the operation is deferred. The Write Control function causes 16 bits to be sent from the processor 40 to interface 70 and sets a control bit in the interface status register if neither the data bit or control bit is already set. If either bit is already set the operation is deferred.

The Read Data Register function causes the contents of a 16-bit data register of the interface 70 to be sent to the processor 40. The contents of the interface data register are loaded by the host computer 20 as either data or control information. The data/control bit of the interface status register is then reset unless neither the data or control bit is set. In that event, the operation is deferred. The Read Status function causes the interface 70 to send to the processor 40 the contents of the interface status register in a format as shown in TABLE XI:

TABLE XI

<u>Position</u>	<u>Meaning</u>
-----------------	----------------

	Bit 0	Data Available From Host Computer
	Bit 1	Control Available From Host Computer
	Bits 7-2	Not Used (Received As Zero)
	Bit 8	Data Available From Processor
	Bit 9	Control Available From Processor
	Bit 10	Valid Parity On Data/Control From Processor To Interface, Byte 0
5	Bit 11	Not Used (Received As Zero)
	Bit 12	Valid Parity On Data/Control From Processor To Interface, Byte 0
	Bit 13	Valid Parity On Data/Control From Processor To Interface, Byte 1
	Bits 15-14	Not Used (Received As Zero)

10 The Read Status function is never deferred. The host
computer 20 communicates with the processor 40 as if it were
an input/output device. Accordingly, the interface to the
host computer 20 is formatted according to standard
input/output commands of the host computer.

15

WHAT IS CLAIMED IS:

1 1. A method for executing at least one high level
2 computer language program, comprising the steps of:
3 encoding at least one statement of said program
4 with at least one token, said token being a variable length
5 bit field have a value representative of at least one
6 semantic element of said statement and a length
7 representative of the context of said semantic element; and
8 executing said token with at least one processor,
9 said executing step comprising branching to at least one
10 instruction or data location according to said value of said
11 token.

1 2. A method according to claim 1, wherein said
2 program is supplied to said processor by a host computer
3 through an interface.

1 3. A method according to claim 2, wherein at least
2 one result of executing said program with said processor is
3 supplied to said host computer through said interface.

1 4. A method for executing at least one high level
2 computer language program, comprising the steps of:
3 encoding at least one statement of said program
4 with a multiplicity of tokens, each of said tokens being a
5 variable length bit field having a value representative of at
6 least one semantic element of said statement and a length
7 representative of the context of said semantic element; and
8 executing said tokens with a single processor, said
9 executing step comprising branching to at least one
10 instruction or data location according to each said value of
11 each of said tokens.

1 5. A method according to claim 4, further
2 comprising the steps of:
3 determining the length in bits of at least one of
4 said tokens;
5 calculating the number of bit positions remaining
6 in at least one fixed-length word of a memory connected to
7 said processor; and
8 packing said token in said word if said length of
9 said token does not exceed said number of bit positions
10 remaining in said word.

1 6. A method according to claim 5, further
2 comprising the steps of:
3 packing a zero value in each bit position remaining
4 in said word if said number of bit positions remaining in
5 said word is less than said length of said token; and
6 packing said token in an adjacent word of said
7 memory.

1 7. A method for executing a high level computer
2 language program, comprising the steps of:
3 encoding said program with a multiplicity of
4 tokens, each of said tokens having a value representative of
5 at least one semantic element of said program;
6 emulating said program, said emulating step
7 comprising branching to at least one instruction according to
8 each said value of each of said tokens; and
9 executing said instruction with a processor, said
10 executing step being performed with said processor in a
11 multiplicity of phases.

1 8. A method according to claim 7, wherein said
2 instruction is a microcode instruction.

1 9. A method according to claim 7, wherein said
2 tokens are ordered in a manner representative of the context
3 of each of said tokens.

1 10. A method according to claim 7, wherein each of
2 said tokens is a variable length bit field having a length
3 representative of the context of said semantic element.

1 11. A method according to claim 7, further
2 comprising the steps of:

3 packing each of said tokens sequentially in at
4 least one fixed-length word of a memory connected to said
5 processor.

1 12. A method according to claim 11, further
2 comprising the steps of:

3 determining the length in bits of each of said
4 tokens;

5 calculating the number of bit positions remaining
6 in said word after packing each of said tokens;

7 packing a zero value in each remaining bit position
8 of said word if said length of a token exceeds said number of
9 remaining bit positions in said word;
10 packing said token in an adjacent fixed-length
11 memory word of said memory; and
12 repeating said calculating and packing steps for at
13 least one succeeding token.

1 13. A method according to claim 7, wherein said
2 program is supplied to said processor by a host computer
3 through an interface.

1 14. A method according to claim 7, wherein at least
2 one result of executing said encoded program is supplied to
3 said host computer through said interface.

1 15. An apparatus for executing a high level
2 computer language program, comprising:
3 a multiplicity of processors, wherein at least two
4 processors are connected to supply signals between said
5 processors;
6 at least one instruction memory connected to at
7 least one of said processors to supply signals representative
8 of at least one instruction, said instruction memory being
9 loadable with at least one microprogram; and
10 at least one cache memory connected to at least one
11 of said processors to supply and receive signals
12 representative of data to be used during execution of said at
13 least one instruction.

1 16. An apparatus according to claim 15, further
2 comprising a main memory connected to at least one processor
3 to supply and receive signals representative of any
4 information to be packed or stored.

1 17. An apparatus according to claim 15, further
2 comprising at least one interface means for receiving and
3 supplying signals between at least two of said processors.

1 18. An apparatus according to claim 15, wherein at
2 least one of said processors executes a multiplicity of
3 instructions in a pipeline manner, each of said instructions
4 in said pipeline being partially executed concurrently with
5 each other instruction in said pipeline.

1 19. An apparatus for executing a high level
2 computer language program, comprising:
3 a single very large scale integrated circuit
4 processor, said processor being contained on a single silicon
5 chip;

6 an instruction memory connected to said processor
7 to supply signals representative of at least one instruction,
8 said instruction memory being loadable with at least one
9 microprogram; and

10 a cache memory connected to said processor to
11 supply and receive signals representative of data generated
12 during execution of said at least one instruction.

1 20. An apparatus according to claim 19, further
2 comprising a main memory connected to said processor to
3 supply and receive signals representative of any information
4 to be packed or stored.

1 21. An apparatus according to claim 19, further
2 comprising an interface means for receiving and supplying
3 signals between a host computer and said processor, said
4 interface means being connected to said processor and said
5 instruction memory.

1 22. An apparatus according to claim 21, wherein
2 said interface means is further connected to said main
3 memory.

1 23. An apparatus according to claim 19, wherein
2 said processor executes a multiplicity of instructions in a
3 pipeline, each instruction in said pipeline being partially
4 executed concurrently with each other instruction in said
5 pipeline.

AMENDED CLAIMS

[received by the International Bureau on 24 May 1988 (24.05.88)
original claims 15, 18 - 23 amended ; other claims unchanged (4 pages)]

7 packing a zero value in each remaining bit
8 position of said word if said length of a token exceeds
9 said number of remaining bit positions in said word;
10 packing said token in an adjacent fixed-
11 length memory word of said memory; and
12 repeating said calculating and packing steps
13 for at least one succeeding token.

1 13. A method according to claim 7, wherein
2 said program is supplied to said processor by a host
3 computer through an interface.

1 14. A method according to claim 7, wherein
2 at least one result of executing said encoded program
3 is supplied to said host computer through said
4 interface.

1 15. An apparatus for executing a high level
2 computer language program, comprising:
3 a multiplicity of processors, wherein at
4 least two processors are connected to supply signals
5 between said processors and at least one of said
6 processors comprises means for executing at least one
7 token, said token being a variable length bit field
8 having a value representative of an operator or operand
9 of said high level computer language program and a bit

10 length representative of a characteristic of said
11 operator or operand, said characteristic being
12 intrinsic to said high level computer language program;
13 at least one instruction memory connected to
14 at least one of said processors to supply signals
15 representative of at least one instruction, said
16 instruction memory being loadable with at least one
17 microprogram; and
18 at least one cache memory connected to at
19 least one of said processors to supply and receive
20 signals representative of data to be used during
21 execution of said at least one instruction.

1 16. An apparatus according to claim 15,
2 further comprising a main memory connected to at least
3 one processor to supply and receive signals
4 representative of any information to be packed or
5 stored.

1 17. An apparatus according to claim 15,
2 further comprising at least one interface means for
3 receiving and supplying signals between at least two of
4 said processors.

1 18. An apparatus according to claim 15,
2 wherein at least one of said processors comprises means
3 for executing a multiplicity of instructions in a
4 pipeline manner, each of said instructions in said
5 pipeline being partially executed concurrently with
6 each other instruction in said pipeline.

1 19. An apparatus for executing a high level
2 computer language program, comprising:
3 processing means for executing at least one
4 token, said token being a variable length bit field
5 having a value representative of an operator or operand
6 of said high level computer language program and a bit
7 length representative of a characteristic of said
8 operator or operand, said characteristic being
9 intrinsic to said high level computer language program;
10 instruction memory means for storing at least
11 one microprogram, said instruction memory means being
12 connected to said processing means to supply signals
13 representative of at least one instruction; and
14 cache memory means for storing data generated
15 during execution of said at least one instruction, said
16 cache memory means being connected to said processing
17 means to supply and receive signals representative of
18 said data.

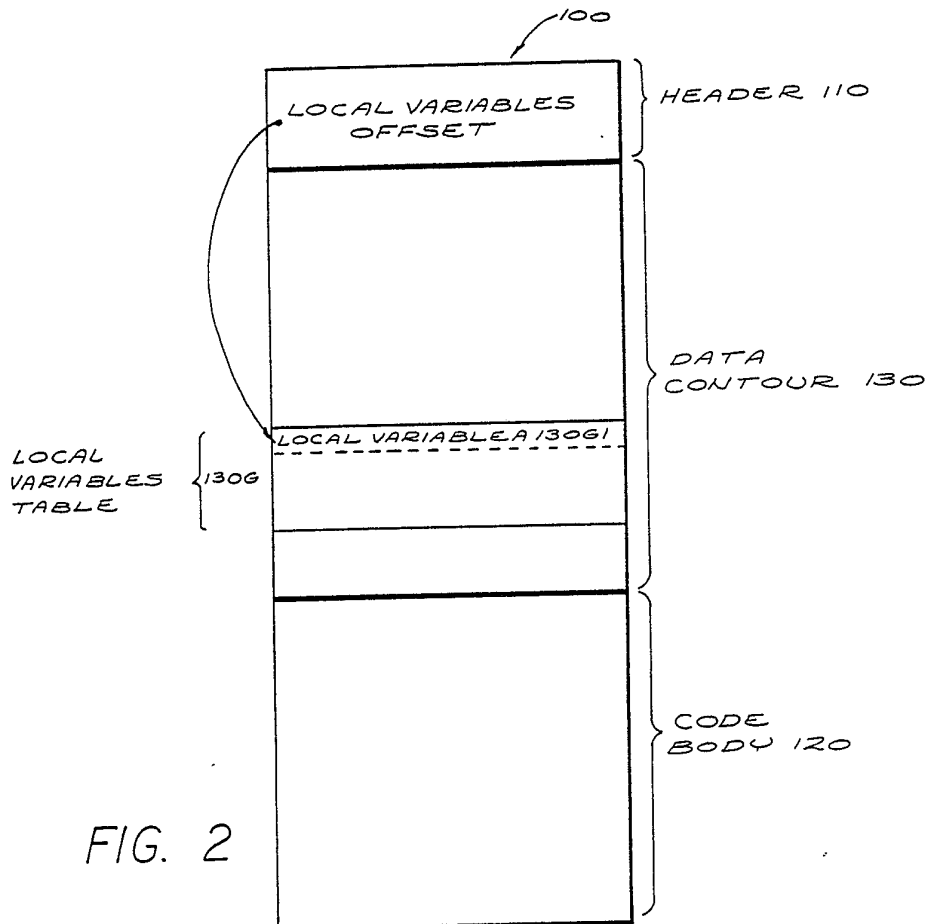
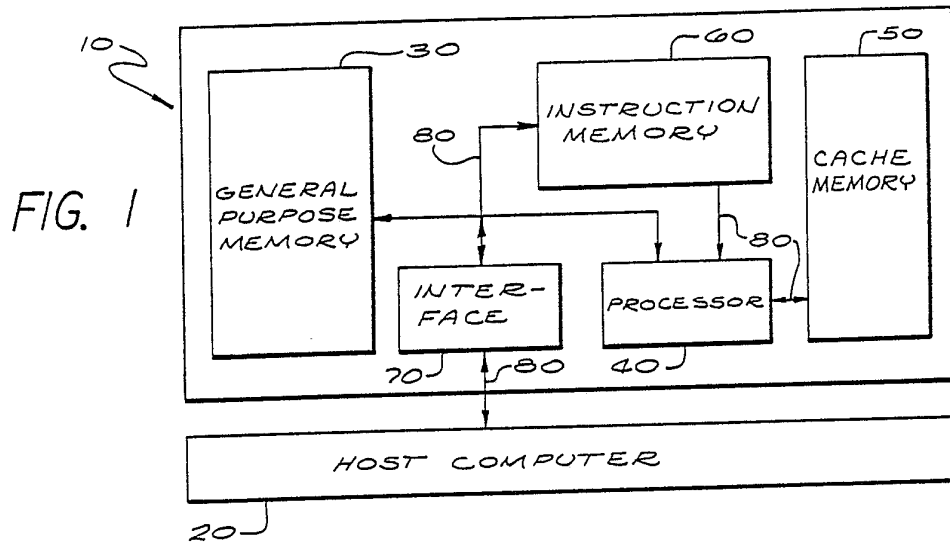
1 20. An apparatus according to claim 19,
2 further comprising main memory means for storing any
3 information to be packed or stored, said main memory
4 means being connected to said processing means to
5 supply and receive signals representative of said
6 information.

1 21. An apparatus according to claim 19,
2 further comprising an interface means for receiving and
3 supplying signals between a host computer and said
4 processing means, said interface means being connected
5 to said processing means and said instruction memory
6 means.

1 22. An apparatus according to claim 21,
2 wherein said interface means is further connected to
3 said main memory means.

1 23. An apparatus according to claim 19,
2 wherein said processing means executes a mulitiplicity
3 of instructions in a pipeline, each instruction in said
4 pipeline being partially executed concurrently with
5 each other instruction in said pipeline.

1/5



2/5

FIG. 3

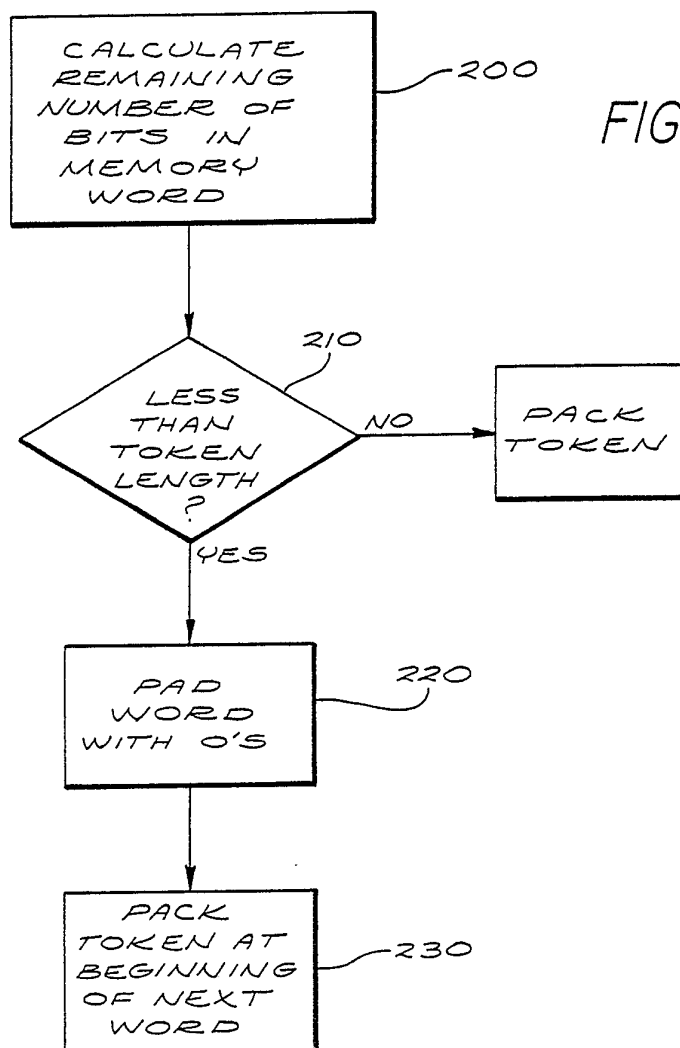


FIG. 4

300
 $A = 10 \div 5$
 310 320 330 340
 {6:1} {2:3} {4:4} {20:20}
 350 360 370
 {2:3} {4:4} {20:10}
 380 390 400 410
 {2:1} {5:1} {2:1} {5:2}
 420 430 440 450 460 470 480
 {2:3} {4:3} {4:1} {2:1} {5:1} {2:1} {5:2}

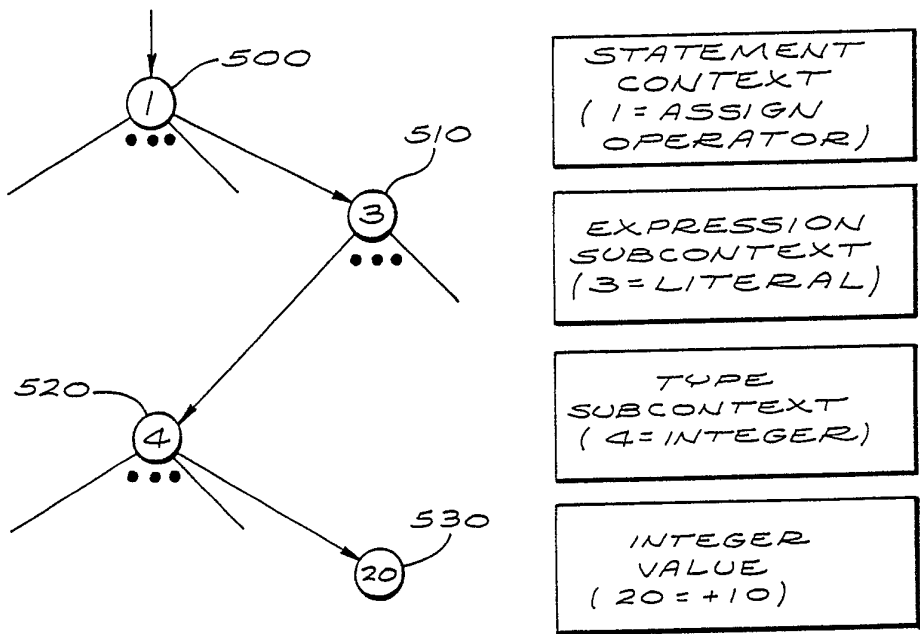


FIG. 5

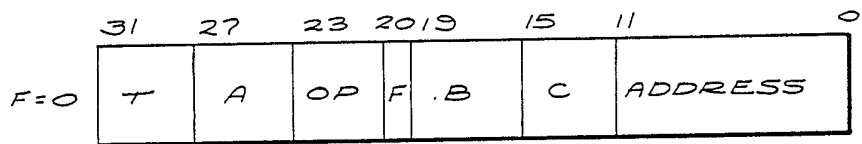


FIG. 6a

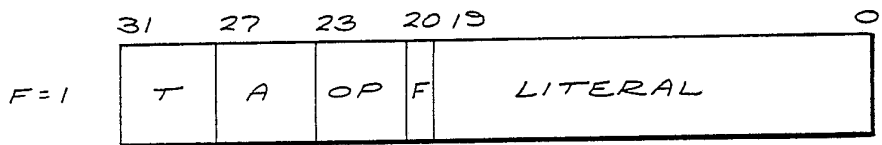
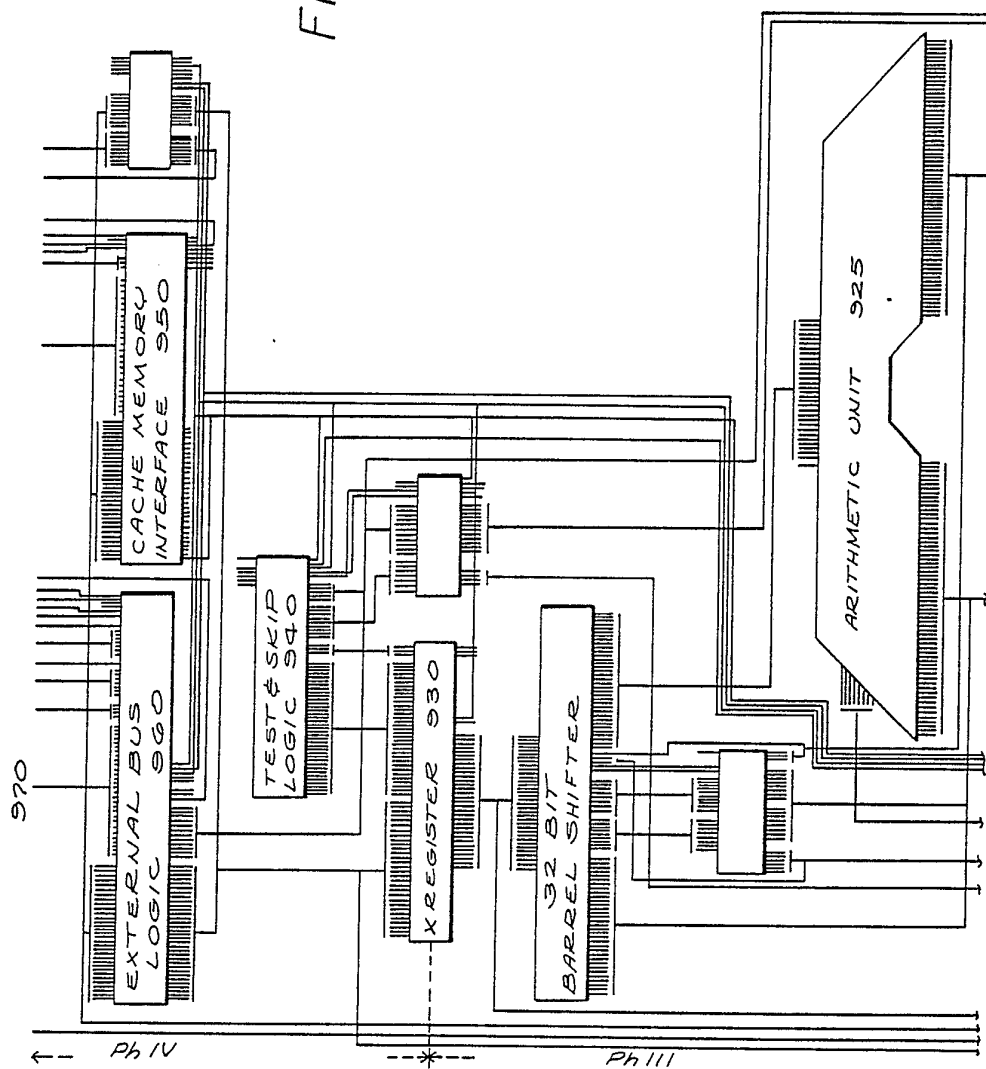


FIG. 6b

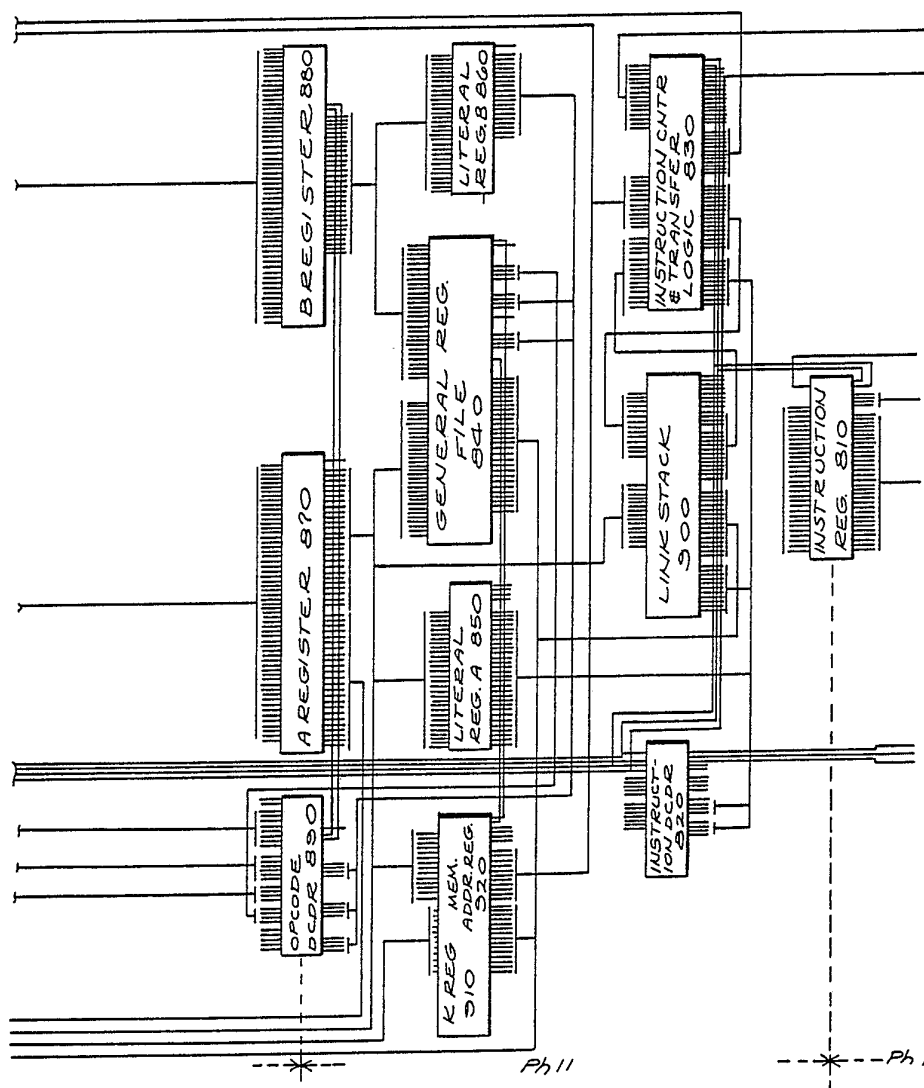
4/5

FIG. 7(1)



5/5

FIG. 7(2)



INTERNATIONAL SEARCH REPORT

International Application No.

PCT/US87/03444

I. CLASSIFICATION OF SUBJECT MATTER (if several classification symbols apply, indicate all) ³		
According to International Patent Classification (IPC) or to both National Classification and IPC IPC (4): G06F 9/34, 9/44 U.S. Cl. 364/200		
II. FIELDS SEARCHED		
Minimum Documentation Searched ⁴		
Classification System	Classification Symbols	
U.S.	364/200, 900 "MSFILE".	
Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched ⁵		
III. DOCUMENTS CONSIDERED TO BE RELEVANT ¹⁴		
Category *	Citation of Document, ¹⁶ with indication, where appropriate, of the relevant passages ¹⁷	Relevant to Claim No. ¹⁸
X	US, A, 4,506,325 (Bennett et al.) 19 March 1985, See ABSTRACT; col. 41, lines 50-65; col. 44, lines 59-60; col. 42, lines 38-39; col. 6, lines 37-70; and col. 7.	1, 3-4, 7-11
Y	US, A, 4,437,184 (Cork et al.) 13 MARCH 1984, See ABSTRACT and Figs 1-2.	2-3, 13-14
X	US, A, 4,456,952 (Mohrman et al.) 26 June 1984, See Fig 4; and ABSTRACT.	15-16, 19
Y	US, A, 4,390,946 (Lane) 28 June 1983, See col. 1, lines 19-35, and col. 3, lines 29-31.	18, 23
EY	US, A, 4,724,521 (Carron et al.) 9 Feb 1988 See col. 7, lines 43-65, and col. 10, lines 50-60.	17, 20-22
A	US, A 4,499,535 (Bachman et al.) 12 Feb 1985 See col. 2.	5-6
¹⁵ * Special categories of cited documents: "A" document defining the general state of the art which is not considered to be of particular relevance "E" earlier document but published on or after the international filing date "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified) "O" document referring to an oral disclosure, use, exhibition or other means "P" document published prior to the international filing date but later than the priority date claimed "T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention "X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step "Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art. "&" document member of the same patent family		
IV. CERTIFICATION		
Date of the Actual Completion of the International Search ²	Date of Mailing of this International Search Report ²	
1 MARCH 1988	23 MAR 1988	
International Searching Authority ¹	Signature of Authorized Officer ²⁰	
ISA/US	Emily Chan <i>Emily Chan</i>	

FURTHER INFORMATION CONTINUED FROM THE SECOND SHEET

V. ☐ OBSERVATIONS WHERE CERTAIN CLAIMS WERE FOUND UNSEARCHABLE ¹⁰

This international search report has not been established in respect of certain claims under Article 17(2) (a) for the following reasons:

1. ☐ Claim numbers because they relate to subject matter ¹² not required to be searched by this Authority, namely:

2. ☐ Claim numbers because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out ¹³, specifically:

VI. ☒ OBSERVATIONS WHERE UNITY OF INVENTION IS LACKING ¹¹

This International Searching Authority found multiple inventions in this international application as follows:

I. Claims 1-14 drawn to a process for executing a high level language classified in class 364 subclass 200.

II Claims 15-23 drawn to an apparatus for executing a high level language classified in class 364 subclass 200.

1. ☒ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims of the international application.

2. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims of the international application for which fees were paid, specifically claims:

3. ☐ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claim numbers:

4. ☐ As all searchable claims could be searched without effort justifying an additional fee, the International Searching Authority did not invite payment of any additional fee.

Remark on Protest

☐ The additional search fees were accompanied by applicant's protest.

☐ No protest accompanied the payment of additional search fees.