EXECUTION MODELS FOR TRANSLATOR DESIGN

BY

MILES T. CLEMENTS JR.

B.S., NORTH GEORGIA COLLEGE, 1965

A MASTER'S REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

KANSAS STATE UNIVERSITY

MANHATTAN, KANSAS

1977

APPROVED BY:

MAJOR PROFESSOR

TABLE OF CONTENTS

Chapter Page

ILLEGIBLE DOCUMENT

THE FOLLOWING DOCUMENT(S) IS OF POOR LEGIBILITY IN THE ORIGINAL

THIS IS THE BEST COPY AVAILABLE

THIS BOOK CONTAINS NUMEROUS PAGE NUMBERS THAT ARE ILLEGIBLE

THIS IS AS RECEIVED FROM THE CUSTOMER

THIS BOOK CONTAINS NUMEROUS PAGES WITH THE ORIGINAL PRINTING BEING SKEWED DIFFERANTLY FROM THE TOP OF THE PAGE TO THE BOTTOM.

THIS IS AS RECEIVED FROM THE CUSTOMER.

Chapter 1

INTRODUCTION

The fundamentals for this report were covered in a Translator Design course taught by Dr.Bill Hankley in the summer of 1975 and summer of 1976. The CS700 stack machine was designed and implemented by the 1975-1976 Translator Design classes. This report combines class presentations and existing CS700 documentation to explain the stack concept and implementation.

An overview of stack machines and the CS700 instruction set is presented to describe the environment. A diagram of an interpreter is shown to graphically show this reports' relationship to other CS700 reports. The tagged architecture is explained in chapter 2 by representing the execution stack as a data structure. Execution models are presented in chapter 3 for selected instructions. Indexing and I/O are not included. A description of a design specification language and formal algorithms for the selected instruction set are included as appendixes.

Stack Machine

A basic stack machine is characterized by module calls which are dynamically linked at the point of call. A given

procedure call may result in further procedure calls in the called procedure,including recursive calls.[1] The general features of the CS700 Interpreter are:

- calls by linkage values(not addresses)
- only activation records,scalars,and address references on the stack.
- activation records, argument linkage, and temporary results on the same execution stack.
- procedure table, static link variables(GLOBALS), symbol table template, and address translation table on another structure (HEAP).
- tagged architecture with dynamic type checking.
- local/global scope rules.
- thirty three instructions in the set

The instruction set for CS700 is shown in table 1.

---

[1] David Gries, Compiler Construction for Digital Computers (New York:John Wiley &Sons,Inc.,1971).

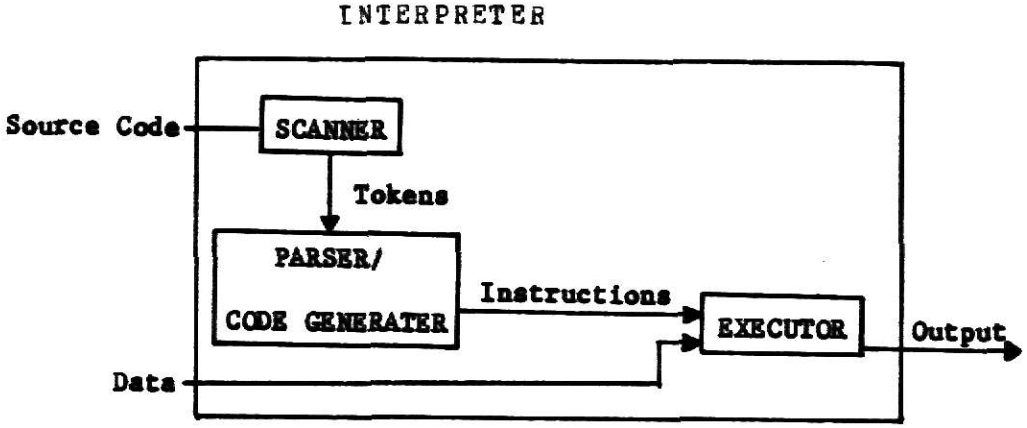
Table 1.

CS700 Instruction Set

	INSTRUCTION	OPERATOR
ASSIGNMENT	ASGN <operand1><operand2>	<=
ARITHMETIC	OPER<operand1>	ceiling,floor,round
		truncate,absolute,sign
	OPER<operand1><operand2>	+,-,/,*,i
RELATIONAL	OPER<operand1><operand2>	=,≠,<,≤,>,≥
LOGICAL	OPER<operand1>	¬
	OPER<operand1><operand2>	∧,∨
CONTROL	OPER<operand1>	BR,BRT,BRF
	OPER<operand1><operand2>	BRI,BRT,BRF
STACK	OPER<operand1>	PUSH,POP
AR	OPER<operand1>	PUSHAR, POPAR
LINK	OPER<operand1>	FLINK,BLINK

Theoretical Framework

A general diagram of an interpreter is shown below:



- Scanner- scans the source code and generates tokens
-Parser- syntactical analyzer
-Code generaters- Generates the instructions
-Executor- Executes the instructions and produces the output.

This report will focus on the execution portion of the interpreter and the CS700 instruction set. The Master's report, Walk-Through of Translator Algorithms, Kansas State Universty, 1977, by James R.Meyer discusses scanning , parse, and code generation.It can be used with this report to obtain an overall understanding of the CS700 Interpreter. **The stack representation presented during the Translator Design classes will be used extensively in this report.**

Essential concepts are presented in chapter 2 and

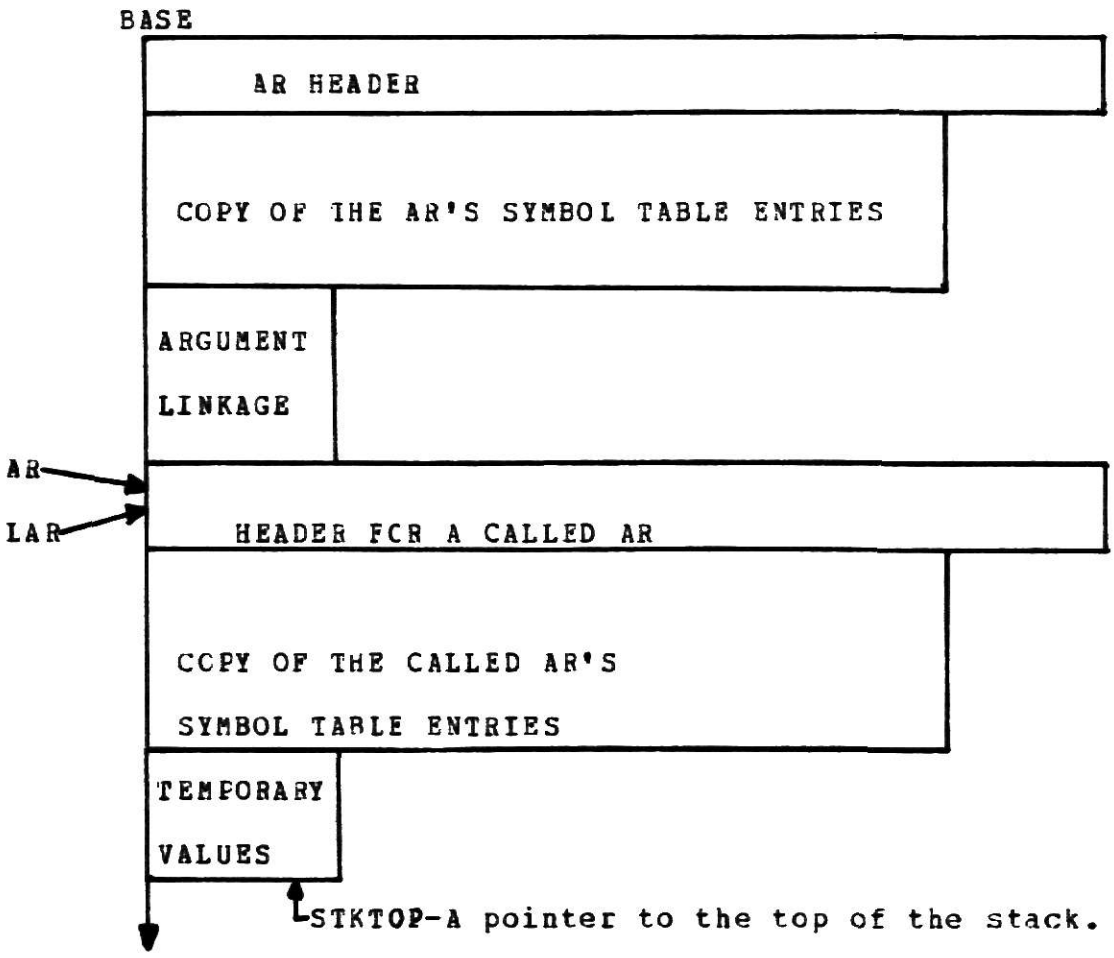
should be understood before reading the remainder of the report. Execution models are presented in Chapter 3. Representative source lines and the resulting generated code are shown and are used as a framework for the execution models. The stack requirements are discussed and snapshots of the execution stack are shown for each instruction type. A sample walk-thru is presented as optional reading in chapter 3 (ASSIGNMENT OPERATOR) to demonstrate the use of the formal algorithms presented in appendix B. The design specification language is described in appendix A. The language notation should be understood before attempting a walk-thru.

The formal algorithms were written to facilitate future enhancements to the CS700 machine and verify the stack representations presented in the report. It must be emphasized that the algorithms are not all inclusive or necessarily error free. The handling of GLOBALS was intentionally omitted since they are handled quite easily but tend to substantially increase the length of the algorithms. This author has walked-thru the algorithms and believes them to be useful as 'spring boards' for class discussions.

Chapter 2

CS700 STACK REPRESENTATIONS

The execution stack consists of activation records(AR), argument linkage from the calling module, and temporaries produced by the current AR. A representation of the stack is shown below:



AR- A pointer to the current AR.

LAR- A pointer to the last AR. In general, LAR=AR. It does not equal AR when a new AR is being added to the stack or is being removed from the stack stack.

Activation Record (AR)

An activation record consists of a header and a copy of the symbol table. The header contains seven subfields which are explained below:

ARTAG	ARLEN	ARLIN	ARINST	ARLCOD	ARPIND	ARRET
AR	7	0	0	2100	7	0

ARRET- The relative index of the previous AR.

ARPIND- Index tc the procedure table.

ARLCOD- Logical address of the code.

ARINST- Relative index for the instruction being executed.

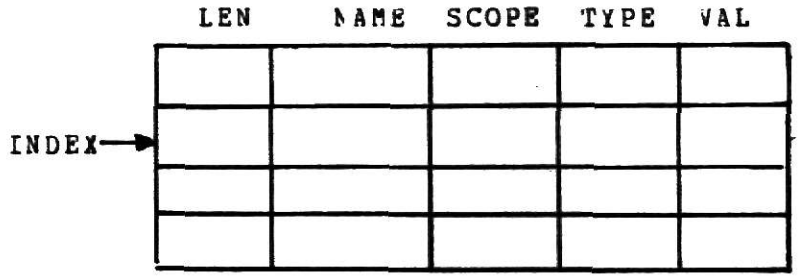
ARLIN- Current source line number being executed.

ARLEN- Length cf the AR.

ARTAG- Tag subfield.A numeric representation for AR.

The entries in the symbol table portion of the AR have five subfields: LEN, NAME, SCOPE, TYPE, and VAL. It is important to note that the first three fields are filled in during scan, parse, and code generation. **The last two subfields** are filled in dynamically during execution. The exception is GLOBAL variables which are static linked during parse and code generation (GLOBAL variables are similar to COMMON variables **in** Fortran). The VAL subfield of GLOBAL entries contains the relative index of the variable in the GLOBAL table. This index , static link, allows the static value to be obtained during execution.

The subfields for symbol table entries are shown below[2]:



INDEX- the relative index for a particular symbol table entry.

LEN- length of variable name

NAME- name of the variable

SCOPE- allowable SCOPE entries are: LOC; IN; OUT; VARY; GLOBAL; and LAB.

SCOPE	MEANING
LOC	Static local variable
IN	Dynamic link variable; TYPE and VAL subfields are passed from the calling AR to the called AR, as arguments.
OUT	Dynamic link variable; TYPE and VAL subfields are returned (copied into) to the calling AR.
VARY	IN or OUT
GLOBAL	Static link variable

TYPE- The following entries are allowable in the TYPE subfield; INTEGER, REAL, BOOLEAN, STRING , or ARRAY.

VAL- The following entries are allowable in the VAL

---

[2] It is noted that the GLOBAL table has the same structure.

subfield depending on the content of the TYPE subfield:

TYPE	VAL subfield content
INT	integer value
REAL	Real value
BOOLEAN	Boolean constant(TRUE or FALSE)
STRING	Address of the string
ARRAY	Address of the array

Argument Linkage

The argument linkage consists of elements having two subfields, TAG and VAL. The content of the argument linkage elements is shown below:

TAG	Content of VAL subfield
VAR	relative index of the variable in the AR.
NARG	number of arguments
INT	integer value
REAL	real value
BOOL	boolean constant
ARRAY	address of the array
STRING	address of the string

A typical example of argument linkage is shown below:

TAG	VAL
INT	5
VAR	4
NARG	2

Temporaries(TEMP)

Temporaries also have two subfields, TAG and VAL. Any of the TAG and VAL subfields used in the argument linkage are allowable as a TEMP except NARG.

CODE TRIPLETS

General form: ⟨Operater⟩ ⟨Operand1⟩ ⟨Operand2⟩

The CS700 instruction set is coded, in general, in triplet form. The three fields composing the instruction are operator , operand1, and operand2. Each field is composed of integer subfields . Symbolic tags are used in this report to represent the code triplets which are used by the CS700 machine. Example:

ADD ⟨VAR,1,3⟩⟨TEMP, - , -⟩ represents ⟨1,4⟩⟨209,1,3⟩⟨208, -, -⟩

The allowable forms of operands are shown below:

OPERAND	MEANING
⟨VAR, index, -⟩	a legal variable
⟨TEMP, - , -⟩	a temporary, top of the execution stack
⟨SCALOR, value, -⟩	a scalar; integer,real,boolean, or string
⟨ARRAY, addr, -⟩	legal address

The operand fields have subfields for TAG, INDEX, and LOC as shown below[3]:

OPD⇒ ⟨TAG, INDEX, LOC⟩

TAG⇒ VAR | CONS | ARRAY | TEMP

CONS⇒ INT | REAL | BOOLEAN | STRING

TEMP⇒ VAR | CONS | ARRAY

INDEX⇒ NINDEX | VALUE

---

[3] The ' | ' is read 'or'. The ' ⇒ ' is read 'reduces to'.

Sample operands and the meaning for each is shown below:

Operand	Meaning
<INT, 5, LOC>	Direct scalar value, 5
<VAR, 3, LOC>	Indirect reference through the AR; the third variable in the AR.
<TEMP, -, ->	Indirect reference through the execution stack.

The operator field has subfields for the operator code and the location of the operator in the source line as shown below:

OP➔ <op_code, loc>

The op_code is an integer code identifying the operation which must be performed. The loc subfield will not be shown in this report in order to simplify the discussions. It is used by the CS70C machine to point to the correct column in the source line if an error occurs.

The INDEX and LOC fields are not always defined. It should be noted that the INDEX and LOC subfields have no meaning when a TEMP is output during parse and code generation. Further, it must be understood that the TAG and INDEX subfields will be defined during execution. Therefore, a TEMP on the execution stack will have TAG and INDEX subfields, but the TAG will not be TEMP. This point will

become obvious during the execution presented in Chapter 3.

Representative code triplets used in the report are shown below.

CODE TRIPLET	MEANING
ADD<VAR, 2, 5><INT, 5, 7>	Add integer 5, found in the seventh column of the source line, to variable 2 in the current AR. Variable 2 occurs in the fifth column of the source line.
ASGN<VAR, 4, -><TEMP, -, ->	Assign the TAG and VALUE subfields of the temporary on the top of the execution stack to variable 4 in the current AR.
ASGN<VAR><VAR|SCALAR|ARRAY|TEMP>	General form of the above instruction. It should be read; operand1 must be a variable, operand2 can be a variable scalar, array, or TEMP.

Chapter 3

EXECUTION MODELS

An execution model, sample execution, is presented and discussed for selected CS700 instructions. In general, source lines have been selected for each model to reinforce previous execution models. Discussions concerning parse and code generation are included as appropriate, especially when the generated code effects the efficiency of the execution. Changing the instruction counter field, ARINST, is common to all of the models. ARINST, the fourth field of the AR header, points to the instruction being executed. Therefore, control is transferred to the next instruction by adding the word length of the current instruction to ARINST.

Representations of the execution stack, snapshots, are presented to give a pictorial view of the stack at any given instant. The following notations are used in the snapshots:

NOTATION	MEANING
//////	Previous activation records and argument linkage.
--	Fields which are not pertinent to a particular discussion or did not change from the previous activation record.

ASSIGNMENT INSTRUCTION

Dyadic operater: ←

General form of the instruction:

OPERATOR<VAR><VAR|SCALAR|ARRAY|TEMP>

The assignment cperater is used tc define or change the type and value fields of the variable specified by operand1. The assignment implies that: an activatien record is on the execution stack; operand2 refers to either a variable fully defined in the activation record or a temporary on the execution stack; operand1 refers to a variable contained in the activation reccrd,tut it's type and value fields need not be previously defined.

Upon assignment,the type and value fields of the variable referenced ty operand1 are erased,if previously defined. The type and value fields of the temporary/variable referenced by operand2 are copied intc the respective fields of the variable specified by operand1. This results in dynamic coersion of type in that cperand1 must accept the type cf cperand2. Consider the source line, Y← X,and the following activation record:

////////

AR	--	--	21	--	--	--
2		P1	PROC	--	--	
--		--	--	--	--	
1		Y	LOC	BOOLEAN	TRUE	
1		X	LOC	INT	5	

A snapshot of the execution stack following execution of ASGN <VAR,3,1><VAR,4,3> would appear as follows:

////////

AR	--	--	29	--	--	--
--		--	--	--	--	
--		--	--	--	--	
1		Y	LOC	INT	5	
1		X	LOC	INT	5	

Changes:

The type field of variable 3 was dynamically changed to integer and the value field was changed to 5.

The instruction counter field was changed to 29.

It should be noted that all variables to the right of an assignment operator must be fully defined (scope,type,and value fields) prior to the assignment,else the assignment results in a run time error. In the above example,variables Y and X were defined in previous source lines.

It should be evident from the above example that the assignment operator,←,is different from the commonly used equal sign ,=, in that assignment can establish or replace the value of the variable on the left of the operator plus cause dynamic coersion of type. Another difference, dynamic allocation is discussed in the ARRAY section of the report.

Formal specifications for the assignment operator are shown in appendix B, figure 1.

Sample Walk-Thru

The previous assignment instruction , ASGN<VAR,3,-><VAR,4,-> and the formal algorithm, figure 1, appendix B. will now be used for a sample Walk-Thru. Line numbers have been added to the algorithm to assist the reader. The snapshot at the top of the previous page shows the condition of the stack when the assignment instruction was encountered. It should be referred to during the Walk-Thru.

LINE	ACTION
2-6	The tag field of operand1 is Var. Likewise the index field passes the assertion. Operand2's tag and index meet the assertion. At this point we should know that the operator and both operands have legal tag fields.

LINE	ACTION
7	Since variable 3's type field is bcolean we fall thru to the case statement.
10	Operand2's tag is not array so we continue to the next case.
16	The tag field is VAR so we must enter.
18	Again the type field of variable 4 causes us to go to the else segment of the IF THEN ELSE syntax.
24	continue
25	Assign variable 4's TYPE field to the type field of variable 3.
26	Place operand2's val field(5) in the VAL field of variable 3.
27	End of the Else
28	End the case,skip to the ENDCASE
34	continue
35	Increase the instruction counter field by the length of the instruction. In this example, we are using 8, so 29 is placed in the instruction counter field.
36	END THE PROCEDURE. It should be noted that we now have the stack condition shown in the previous snapshot.

MONADIC ARITHMETIC INSTRUCTIONS

Operators: ceiling, floor, round, truncate, absolute, sign

General form of instructions: operator<var/value/temp>

The six monadic arithmetic operators presently implemented in CS700 are shown below, each accompanied by its mathmatical meaning:

OPERATOR	CS700 NOTATION	MEANING
Ceiling	⌈	Return the smallest integer greator than or equal to operand. For example, -7.6 becomes -7 , 7.6 becomes 8.
Floor	⌊	Return the largest integer less than of equal to operand. For example, -7.6 becomes -8, 7.6 becomes 7.
Round	RD	Chop(operand+0.5). For example, -7.6 becomes -8, 7.6 becomes 8.
Truncate	TR	Operand is truncated at the decimal point. For example, -7.6 becomes -7 , 7.6 becomes 7.
Absolute	ABS	Absolute value of operand, |operand|.

For example -7.6 becomes 7.6, 7.6 remains 7.6.

Sign SIGN

Return a -1 if operand is less than zero, 0 if operand is zero, or 1 if operand is greater than zero. For example, -7.6 becomes -1, 7.6 becomes 1, and 0 remains 0.

Each of the above operators operate on two data types,integer or real. Arguments in function references may be constants, variables,or expressions. For example, X← ABS(-5), X ← ABS(Y) ,or X ← ABS(Y -5).

CS700 functions are intended to operate on arrays. The functions operate upon each element of the array, and return an array of results of the same dimension as the argument.

Monadic arithmetic instructions imply that an activation record is on the execution stack and the type and value fields are specified within the operand; by a variable in the symbol table portion of the current activation record; or by a temporary on the top of the execution stack.

The arithmetic operators result in a a temporary value being pushed on the execution stack. For example,consider the source line, Y ← Γ X, and the following activation record:

////////

AR	--	--	30	--	--	--
2		--	PRCC	--	--	
1		Y	LCC			
1		X	LOC	REAL	7.3	

A snapshot of the execution stack following execution of CEIL<VAR,3,4> is shown below:

////////

AR	--	--	35	--	--	--
--		--	--	--	--	
1		Y	LCC			
1		X	LOC	REAL	7.3	
INT	8					

Changes:

1- A temporary resulting from the ceiling operation was pushed onto the execution stack.

2- The instruction counter field was changed to 35.

The next instruction might be an assignment, ASGN<VAR,2,1><temp,-,->, which would cause the temporary to be copied into the type and value fields of variable 2. It should be noted that operand2 of the assignment instruction indicates temporary since the actual type is not known at parse and code generation time.

Formal algorithms for the monadic arithmetic instructions are shown in appendix B , figure 2.

DYADIC ARITHMETIC OPERATERS

Operators: + , - , * , / , ↑

General form: Operator<VAR/TEMP/VAL><VAR/TEMP/VAL>

The dyadic arithmetic instructions imply that an activation record is on the execution stack and both operands are fully defined. The fields can be defined in the symbol table portion of the current activation record; in the global table;a temporary on the top of the execution stack;or in the operand.

The dyadic arithmetic operators are the means by which computations involving addition,subtraction,multiplication, division,and exponentiation are performed. Type checking and conversion,if required,is performed dynamically by the operation.

Each arithmetic instruction results in a temporary being pushed on the execution stack. If both operands are real or if either is real,the resulting temporary will be real. If both operands are integer the resulting temporary will be an integer.

Consider the following symbol table:

4	MAIN	PROC	--	--
1	Y	LOC		
1	X	LOC		

The fifth source line ,X←Y+5, would cause the following code to be generated during code generation:

INSTRUCTION	CODE
19	PUSH<LINE=5>
20	ADD<VAR,2,5><INT,5,7>
28	ASGN<VAR,3,1><TEMP,-,->
36	POP

During execution ,a snapshot of the execution stack could appear as follows following execution of the PUSH instruction:

////////

AR	--	5	20	--	--	--
-		--	Proc	--	--	
1		Y	LOC	REAL	7.3	
1		X	LOC			

Changes: The source line field was changed to 5 and the instruction field was changed to 20.

Execution of the above ADD instruction would result in the following snapshot:

////////

AR	__	5	28	--	--	--
-		--	--	--	--	
-		--	--	--	--	
-		--	-	--	--	

REAL 12.3

Changes: 1- A temporary was pushed on the execution stack.

2- The instruction field was changed to 28.

Execution of the assignment instruction would cause variable 3's type field to be defined REAL and its value field to be defined 12.3. The POP instruction would cause the temporary, REAL 12.3 ,to be effectively erased,since the pointer for the top of the stack(STKTOP) would be decremented.

Execution of other DYADIC arithmetic instructions would be the same as shown for the ADD instruction,except the specified computation would be performed. **The formal algorithms for the dyadic arithmetic instructions are shown in appendix B, figure 3.**

RELATIONAL INSTRUCTIONS

OPERATORS: =,≠,>,≥,<,≤

General form: OPER<VAR/TEMP/VAL><VAR/TEMP/VAL>

Relational operators are the means by which either a TRUE or FALSE relationship is established between two arithmetic expressions. There are six relational operators , each of which is shown below accompanied by its mathmatical meaning:

CS700 NOTATION	MEANING
=	equal to
≠	not equal to
>	greater than
≥	greater than or equal to
<	less than
≤	less than or equal to

The relational instructions result in a BOOLEAN temporary being pushed on the execution stack. The integer code for boolean is placed in the type field of the temporary. A binary one '1'B, in the value field represents TRUE and a binary zero,'0'B,in the value field represents FALSE.

The relational instructions imply that an activation record is on the execution stack and both operands are

'fully defined' as discussed under DYADIC arithmetic operators.

Consider the sixth source line, X←Y≤J, and the following instructions:

INSTRUCTION	CODE
29	PUSH<LINE=6>
30	LE<VAR,2,3><VAR,3,5>
38	ASGN<VAR,4,1><TEMP,-,->
46	POP

The execution stack could appear as follows prior to execution of the relational instruction:

////////

AR	-	6	30	--	--	--
-		--	PROC	--	--	
1		Y	LOC	INT	5	
1		J	LOC	INT	3	
1		X	LOC			

A snapshot of changes in the execution stack following execution of the relational instruction is shown below:

```
////////
AR    --        --        38        --        --        --
 -                --        --        --        --
 -                --        --        --        --
 -                --        --        --        --
 -                --        --        --        --
BOOL FALSE
```

The assignment instruction will copy the type and value fields of the temporary boolean into the respective fields of variable 4 and change the instruction field to 46. The POP instruction would pop the temporary from the execution stack ,by decrementing STKTOP and changing the instruction field to 47.

All the relational instructions would be executed in the same manner as the LE instruction shown above except the temporary pushed on the execution stack would be the result of the specified relational operation. **The formal algorithms for the relational instructions are shown in appendix B, figure 4.**

LOGICAL INSTRUCTIONS

Monadic operator: ¬

Dyadic operator: ∧ , ∨

General form: Monadic operator<temp/val/var>

dyadic operator<temp/val/var><temp/val/var>

There are three logical operators which cause the evaluation of the truth or falsity of logical expressions. Two of the three operators are the connectives AND(∧) and OR(∨),and the third logical operator is the inverter NOT (¬). Each of the logical operators result in either Boolean True or Boolean False being pushed on the execution stack.

The connective,AND,requires the logical expression both preceding and following it be TRUE for the resultant PUSH to be TRUE. if either is FALSE or both are FALSE then the resultant PUSH will be FALSE.

The connective ,OR, requires the logical expression either preceding and/or following it be TRUE for the resultant PUSH to be TRUE. If both are FALSE,then Boolean False is pushed on the execution stack.

The inverter NOT causes the resultant PUSH to be TRUE if the immediately following logical expression is FALSE. Therefore NOT is a monadic operator whereas AND and OR are dyadic.

Consider a fourth source line, A←X∧¬Y , and the following symbol table entries:

--	--	PRCC		
1	X	LCC		
1	Y	LCC		
1	A	LCC		

The following instructions would have been generated during parse and code generation:

INSTRUCTION NUMBER	CODE
20	PUSH<LINE=4>
21	NOT<VAR,3,6>
26	AND<VAR,2,3><TEMP,-,->
34	ASGN<VAR,4,1><TEMP,-,->
42	POP

It should be noted that the variables referenced in a logical instruction must be Boolean type and the value field must contain either of the logical constants, TRUE or FALSE, else a run time error occurs. A snapshot of the execution stack following execution of the PUSH instruction could appear as follows:

////////

AR	--	4	21	--	--	--
2		P2	PROC	--	--	
1		X	LOC	Boolean	True	
1		Y	LOC	Boolean	True	
1		A	LOC			

Execution continues as follows:

INSTRUCTION	RESULT OF EXECUTION
21	Boolean False is pushed on the execution stack. Instruction counter field is changed to 26.
26	The temporary Boolean False, on the execution stack is examined and popped. Type checking is performed. The AND operation is perfcrmed resulting in Boolean False being pushed on the execution stack. The instruction counter is changed to 34.
34	Boolean is copied into variable 4's type field .False is copied into the value field. The instruction counter is changed to 42.

The POP instructicn pops the temporary from the executicn stack and increments the instruction counter.

Formal specifications for the logical instructions are shown in appendix B, figure 5.

CCNTROL INSTRUCTIONS

Control instructicns provide the computer with the capability to choose between two or more sequences of instructions during execution of a program. Control can be transferred by a BRANCH instruction or an INDIRECT BRANCH instruction.

The BRANCH instruction implies that the operand is a value and the value identifies the instruction number. The branch is direct to the specified instuction number.

The INDIRECT BRANCH instruction implies that the operand is a variable. The branch is indirect thru the symbol table portion of the current activation reccrd. The instruction number is obtained from the value/address field of the specified variable.

Transfer of ccntrol can be conditional or unconditional. Conditional transfer of control implies evaluation of a Boolean which must exist on the top of the execution stack. The general format for control instructions and the meaning for each is shown below:

Branch(BR) Instructions	Meaning
BR<value>	Unconditional branch to the instruction number specified by value.
BRF<value>	Branch to the instruction number specified by value if the temporary on the top of the stack is Boolean False.
BRT<value>	Branch to the instruction number specified by value if the temporary on the top of the stack is Boolean True.

INDIRECT BRANCH(BRI)

Instructions	Meaning
BRI<variable>	Unconditional indirect branch to the instruction number.
BRF<variable>	Indirect Branch to the instruction number if the temporary on the top of the stack is **Boolean False.**
BRT;variable>	Indirect Branch to the instruction number if the **temporary** on the top of the stack is Boolean **True.**

In single pass compilers, BRANCH instructions are used for backward branches to previously identified labels or when the operand containing the instruction number can be patched easily. For example: a direct branch on false instruction BRF, is pushed when the THEN of IF THEN ELSE syntax is recognized. The operand will be filled in, patched, when the ELSE is encountered. A patching example will be shown in the next sample execution. It should be noted that BRANCH instructions give more efficient execution than INDIRECT BRANCH instructions , since a look into the symbol table portion of the current activation record is not required during execution.

INDIRECT BRANCH instructions are used to branch forward to a label which is not fully defined at code generation time. For example: consider the source code, GO TO JOE . If the symbol table entry for JOE's address field is null, then an INDIRECT BRANCH, BRI, instruction must be generated. This implies that the symbol table entry for JOE's address field will be entered in the symbol table as a result of compiling subsequent lines of source code. It should be noted that if the label, JOE, had been fully defined, then a direct BRANCH, BR, could have been generated since the address field would contain the correct instruction number.

The following sample code generation and execution demonstrates the use of BRANCH and INDIRECT BRANCH instructions: consider the following line of source code

and symbol table: IF(J= 1) THEN (X=Y) ELSE GO TO JIM

-	-	PROC
1	J	LOC
1	Y	LOC
1	X	LOC
3	JIM	LABEL

Assuming the above line of code was the tenth source line and 28 words had been output as code previously,then the following code would be output during code generation:

INSTRUCTION	CODE
29	PUSH<LINE= 10)
30	EQ<VAR,2,4)<INT,1,7)
38	BRF<INT,51,10)
43	ASGN<VAR,4,16)<VAR,3,18)
51	BRI<VAR,5,21)

THEN in the source line resulted in a direct BRANCH on FALSE,BRF,being generated. The '51' in the operand of the BRF instruction was initially null.Instruction number 38 was pushed on a patch stack,which indicated that the operand had to be patched when the ELSE was encountered.

ELSE caused the patch stack to be popped and the current instruction number,51,to be patched into the operand of instruction 38. Further ,an unconditional INDIRECT BRANCH, BRI,was generated since since JIM's address field was null. The BRI instruction,51, implies that during

execution control will be transferred to the value/address field of variable 5.It should be noted that a compile time error will occur if the address field of variable 5 remains null.

A snapshot of the execution stack following execution of instruction number 29 could appear as follows:

////////

AR	--	10	30	--	2100	--
2		P2	PROC	--	--	
1		J	LOC	INT	4	
1		Y	LOC	REAL	7.2	
1		X	LOC			
3		JIM	LABEL	INT	70	

Changes:The instruction counter changed to 30.

Execution of instruction 30 would result in a Boolean False being pushed on the execution stack and the instruction counter being changed to 38.

Execution of instruction 38 would cause the temporary boolean to be evaluated .Since it is false the instruction counter is changed to 51,a direct BRANCH to instruction 51.

Execution of instruction 51 results in the instruction counter being changed to 70,an INDIRECT BRANCH to the value/address field of variable 5.

It should be noted that the logical address for the instruction being executed is obtained by adding the logical address of the code, ARLCOD(2100), to ARINST(30, 38, 51, 70, etc.).

The formal algorithms are contained in appendix B, figure 6.

STACK INSTRUCTIONS

There are two stack instructions, PUSH and POP, used to manipulate the temporary values and argument linkage of the execution stack. The PUSH instruction is used to update the AR header, and place temporary values and argument linkage on the execution stack. The POP instruction is used to delete temporary values from the execution stack.

PUSH INSTRUCTION

Each new line of source code results in a PUSH instruction in order to update the program line number in the current activation record. The program line number, ARLIN, is the third field in the header of the current AR.

A call statement commonly results in a series of PUSH instructions in order to keep the line number current and link the calling arguments into a new activation record. Example: consider the following symbol table:

-	--	PROC
1	X	LOC
1	Z	LOC
2	P2	PROC

Program line 4 contains the source code CALL P2(X,Z),which results in the following instructions being output at code generatien time:

INSTRUCTION	CODE
8	PUSH<LINE=4>
9	PUSH<VAR,2,->
14	PUSH<VAR,3,->
19	PUSH<NARG,2,->
24	PUSHAR<VAR,4,->

PUSH<LINE=4> updates the source line number,the third field,in the header of the current AR.

AR HEADER before execution:

		ARLIN	ARINST			
AR	--	3	8	--	--	--

AR HEADER after execution:

AR	--	4	9	--	--	--

Changes:The ARLIN field was changed to 4.The ARINST field was incremented by one since a full word is used for this instruction as currently implemented in CS700.

It should be noted that each source line results in a PUSH instruction to update the ARLIN field,whereas each instruction causes the ARINST field to be changed.

Instructions 9-24 effectively push the required linkage

on the execution stack as shown below:

Execution stack following execution of PUSH<VAR,2,->

////////

AR	--	4	14	--	--	--
2		--	PROC	--	--	
1		X	LOC	--	--	
1		Z	LOC	--	--	
2		P2	PROC	--	--	
VAR	2					

Changes:

1. <VAR,2> was pushed onto the execution stack. It should be noted that <VAR,2> refers to X, the second variable in the current AR,which was the first argument in the call.

2. The ARINST field was changed from 9 tc 14.

Execution of the next two PUSH instructions would produce the following snapshot of the execution stack:

////////

		ARLIN	ARINST			
AR	--	4	24	--	--	--
--		--	--	--	--	
--		--	--	--	--	
--		--	--	--	--	
--		--	--	--	--	
--	-					
VAR	3					
NARG	2					

Program line 4 is still being executed. The ARINST field was changed to 19 and 24 as each of the instructions was executed, since the PUSH instruction requires five words as implemented in CS700.

The number of arguments in the call is represented by NARG 2 ,which will be used to formally link the arguments into the called AR. This formal linking and execution of the PUSHAR instruction will be discussed separately.

It should be noted that during execution of arithmetic, relational, and logical instructions, the PUSH instruction is used to place temporaries on the execution stack for future reference.

POP INSTRUCTION

The POP instruction effectively deletes the temporary on the top of the execution stack by decrementing the pointer, STKTOP. It should be noted that any reference to a temporary, except an assignment, results in it being popped from the execution stack as implemented in CS700.

AR INSTRUCTIONS

Two instructions,PUSHAR and POPAR, are used to PUSH and POP activation records.The PUSHAR instruction transfers control to the called program by pushing the new AR on the execution stack. The POPAR instruction returns control to the calling program by poping the current AR.

PUSHAR INSTRUCTION

General form: PUSHAR<VAR,NINDEX,LOC>

An activation record consists of a header and a copy of the symbol table.Therefore,the PUSHAR instruction must:initialize the header;copy the symbol table onto the execution stack;and update the stack pointers.

Initialization of a sample AR header is shown below:

ARTAG	ARLEN	ARLIN	ARINST	ARLOCD	ARPIND	ARRET
AR	7	0	0	2100	7	0

ARRET- The relative index of the previous AR.This subfield is used to return control to the calling AR when a POPAR instruction is encountered.The '0' shown above indicates that this is the main program.For other

programs, AR.ARRET=LAR ,where LAR is the relative index of the current AR.

ARPIND- Index to the procedure table.

ARLCOD- Logical address of the code.This is copied from the procedure table.It should be noted that the logical address of the current instruction equals ARLCOD+ARINST.

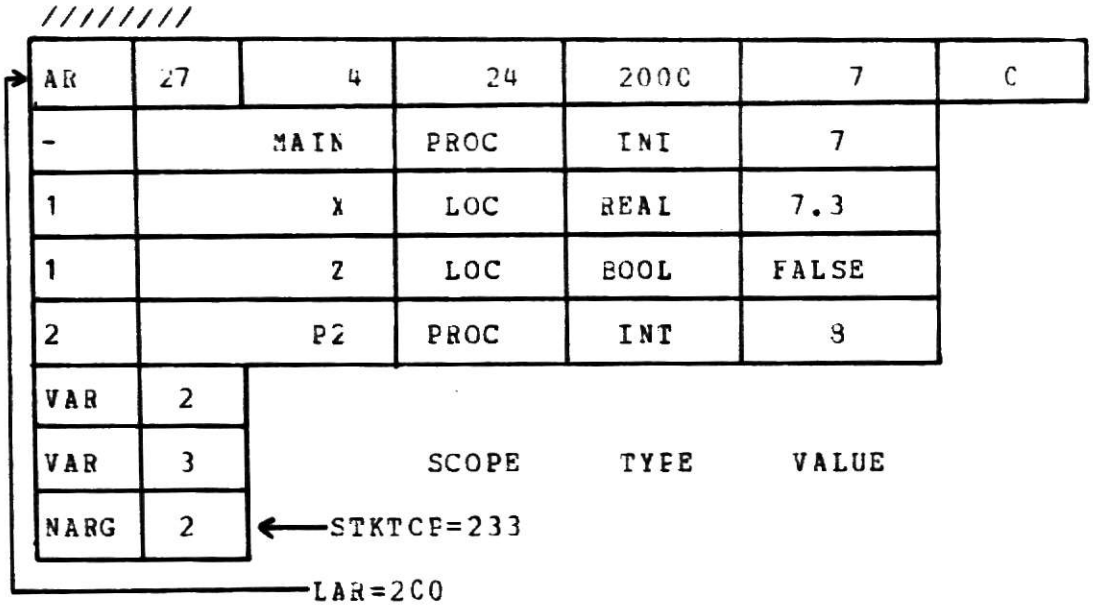
ARINST- Relative index for the instruction being executed.

ARLIN- Current source line number being executed.

ARLEN- Length of the AR.Initially AR.ARLEN=7, since the header requires 7 words.It is updated when the symbol table is copied,AR.ARLEN=AR.ARLEN+LENGTH OF THE symbol table.

ARTAG- Tag subfield.A numeric representation for AR.

The formal specifications for the PUSHAR instruction is shown in appendix B, figure 8. A snapshot of a typical execution stack prior to executing a PUSHAR instruction is shown below:



Consider execution of the following instruction:

PUSHAR<VAR,4,->

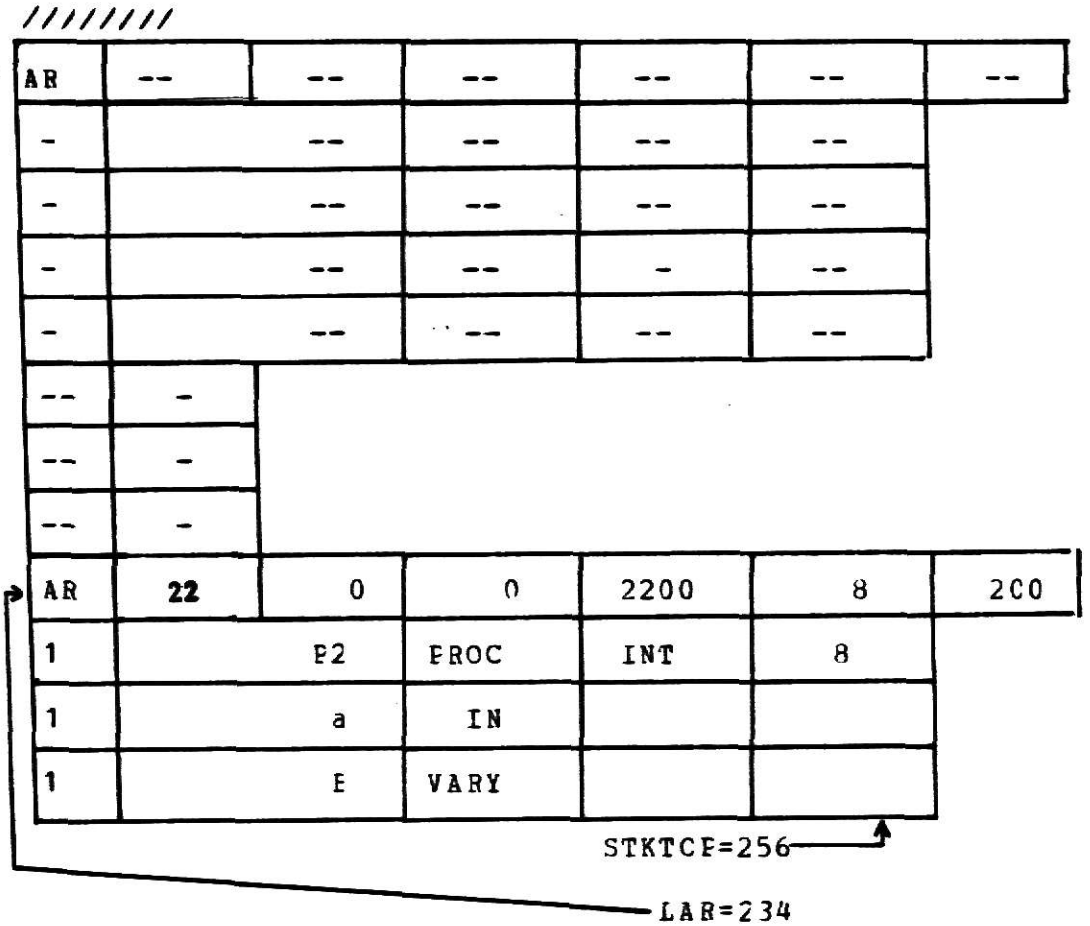
The instruction implies that:

1- The scope field of the variable named in the operand is PROC.

2- The called AR, variable 4,is fully defined in the procedure table. The value field of variable 4 contains PINDEX,the index into the procedure table.

3- The required argument linkage is on the execution stack. NARG,the number of arguments,must be on the top of the stack. Temporaries,representing the arguments must precede NARG.

Execution of PUSHAR<VAR,4,-> would result in the following snapshot of the execution stack:



Changes:

1- The header for the new AR was created.

2. The symbol table for P2 was copied cnto the execution stack.

3- The stack pointers,LAR and STKTCP,were updated.

It should be noted that control has been transferred to P2.The next instruction to be executed will be the first instruction of P2.The address is found at locatien 22C0 in **the address translation table.**

POPAR INSTRUCTION

General form: POPAR

The POPAR instruction returns control to the calling AR by moving the stack pointer, STKTOP and LAR, which point to the top of the stack and the current activation record respectively. When the pointers have been moved, the instruction counter is updated which transfers control to the next instruction to be executed.

The formal specifications for the POPAR instruction is shown in appendix B, figure 8. A POPAR instruction implies that:

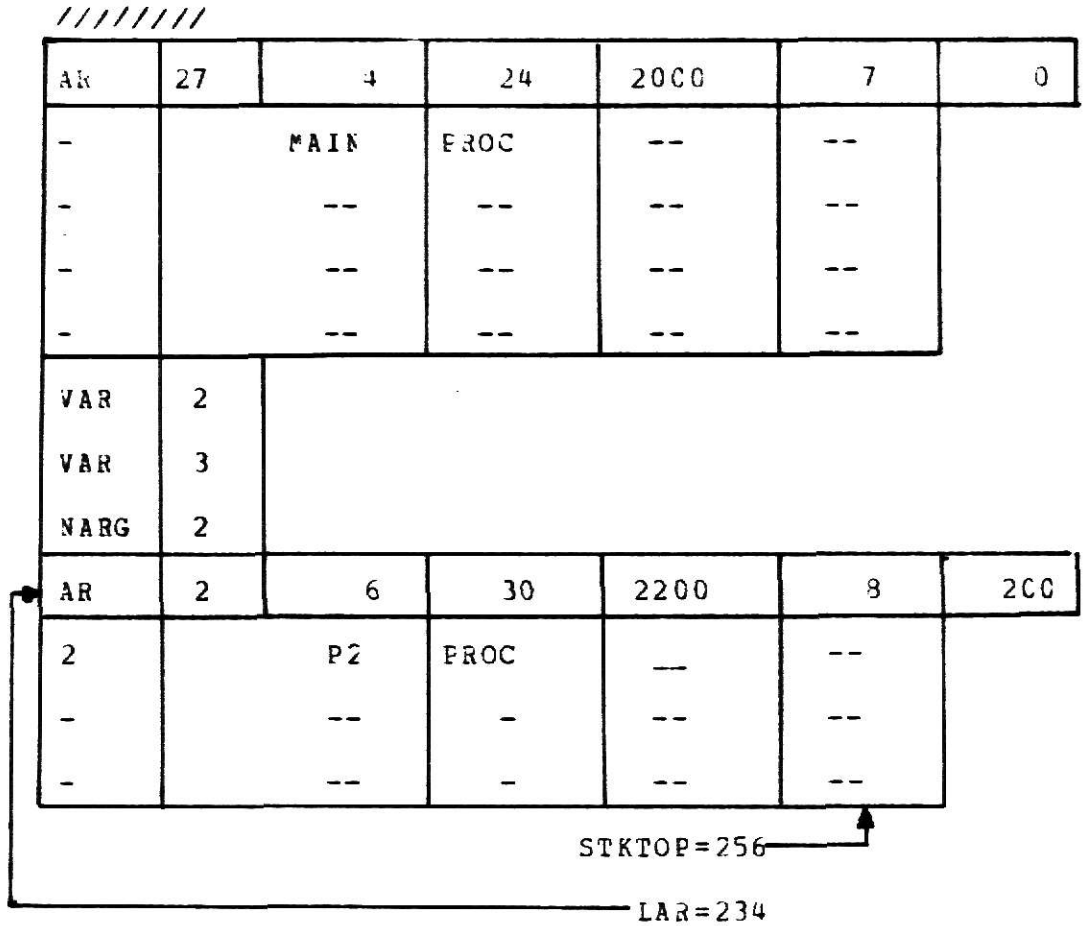
1- Return arguments have already been linked into the calling AR.

2- Control is to be returned to the calling AR, which begins at the logical address specified by AR.ADRET.

3- The argument linkage and anything following it can be deleted.

4- A PUSHAR instruction was the last instruction executed by the calling AR, therefore the instruction counter must be increased by the length of the instruction to transfer control to the next instruction.

A snapshot of a typical execution stack prior to execution of a POPAR instruction is shown below:



Execution of POPAR would result in the following snapshot of the execution stack:

```
//////
AR     27        4        29       2000        7         0
4              MAIN     PROC       --        --
-              --       --         --        --
-              --       --         --        --
-              --       --         --        --
                                STKTOP=227
                                   LAR=200
```

Changes:

1- STKTOP was moved to the last entry preceding the argument linkage.

2- **LAR** was moved to the logical address

specified by AR.ARRET,which makes MAIN the current AR.

3- AR.ARINST was increased by the length of the PUSHAR instruction(24 to 29).

LINK INSTRUCTIONS

The two link instructions,FLINK and BLINK,are used to link calling arguments into a new AR and return arguments to the calling AR respectively.

FORWARD LINK INSTRUCTION

General form: FLINK<VAR,NINDEX,LOC>

The forward link instruction is used to link calling arguments into the called AR,program. Forward links imply that:

1- Both the calling and called AR are on the execution stack.

2- The argument linkage immediately precedes the called AR on the execution stack. The argument linkage contains the number of arguments,NARG, and either the values or references to the variables which are to be linked into the called AR.

3- The first argument in the linkage corresponds to the first argument in both the calling and called AR, etc. Each FLINK instruction has a corresponding argument in the linkage.

4- The arguments in the linkage refer either to a variable in the previous AR or a constant,whereas the operand of the FLINK instruction refers to a variable in the current AR.

5- The scope of variables contained in the operand of the FLINK instruction must have been defined as either IN or VARY.

6- If the value/address field of the variable,which is to be linked into the new AR,contains an address then a copy must be made of the object and the new address linked into the new AR. Therefore,a copy must be made if the type is ARRAY.

Formal specifications for the FLINK instruction is contained in appendix B, figure 9. A snapshot of an execution stack is shown below prior to execution of a FLINK instruction:

```
////////
AR    --      --      --      --      --      --
4           MAIN    PROC      --      --
-           --      --      --      --
1             A     LOC       INT      5
1             B     LOC     ARRAY     500
VAR   3
VAR   4
NARG  2
AR    --      1       7       --      --      --
2           P2      PROC      --      --
1             X       IN
1             Z     VARY
```

Changes in the execution stack are shown in the following snapshot following execution of FLINK<VAR,2,-> :

```
////////
AR     --        --        12        --        --        --
2                P2      PROC        --        --
1                X         IN       INT         5
-               --        --
```

Changes:

1 - Since NARG=2,the first argument which is to be linked begins four words prior to NARG. Therefore,variable 3 in the previous AR must be linked **into** variable 2 of the current AR.

2- The type and value fields of A,the third variable in the previous AR are copied into the respective fields of X, the second variable of the current AR.

3- Control is transferred to the next instruction by increasing PRINST by 5,since the FLINK instruction is 5 words long.

changes in the execution stack following execution of FLINK<VAR,3,-> are shown in the following snapshot:

```
////////
AR     --        --        17        --        --        --
-               --        --        --        --
-               --         -        --        --
-               --        --        --        --
1                Z       VARY      ARRAY      520
```

Changes:

1- ARRAY, the type field of the fourth variable in

the previous AR is copied into the type field of the third var variable in the current AR.

2- Type field,ARRAY, implies that the value/address field contains an address. Therefore ,the header for the array is examined for the size of the array and the proper amount of space is allocated in the HEAP for a copy of the array. The NREF field of the array header equals 1 to reflect one reference to the array.The array is then copied into an address beginning at 520. 520 is placed in the address field of variable 3 in the current AR.

3- The instruction counter is changed to 17.

BACKWARD LINK INSTRUCTION

General form: BLINK<VAR,NINDEX,LCC>

The backward link instructions are used to link arguments of a called AR back into the calling AR. Backward link instructions in effect return values to the calling AR. They imply that:

1- The variable referenced in the operand of a BLINK instruction is contained in the current AR.

2- The scope of any variable appearing in the BLINK operand must be either OUT or VARY.

3 - The argument can be correctly linked to the previous AR after examining the argument linkage,since all calling arguments were pushed.

4- NINDEX in the BLINK operand can be used to identify the corresponding argument in the linkage,since NINDEX minus one identifies the argument placement. NINDEX=3 implies the second argument in the linkage,etc.

5- Both the type and value/address fields are to be copied from the current AR to the previous AR.

6- If the value/address field of the variable in the recieving AR contains an address,then the NREF field of the object header must be decremented. If NREF=0,the space can be deallocated.

The formal specifications for the BLINK instruction is presented in appendix B, figure 9. A snapshot of an execution stack prior to executing a BLINK instruction is shown below:

```
////////
AR    --      --      --      --      --      --
4             MAIN    PROC    --      --
-             --      --      --      --
1               A     LOC     INT      5
1               B     LOC     ARRAY   500
VAR   3
VAR   4
NARG  2
AR    --        4     30      --      --      --
2               P2    PROC    --      --
1               X     IN      INT      5
1               Z     VARY    ARRAY   560
```

Changes in the execution stack following execution of BLINK<VAR,3,-> are shown in the follcwinq snapshot:

```
////////
AR     --      --      --      --      --      --
4              MAIN    PROC      --      --
-              --      --        --      --
1               A      LOC       INT      5
1               B      LOC       ARRAY    560
VAR    3
VAR    4
NARG   2
AR     --      --      35        --      --      --
2              P2      PROC      --      --
1               X      IN        INT      5
1               Z      VARY      ARRAY    560
```

Discussion:

1-NINDEX=3 **implies** that the receiving variable is identified by the second argument in the linkage.Therefore variable 4 in the previous AR was examined.Since it's type field was array,NREF in the array header was decremented and in this case the block referenced by the 500 was deallocated.

2- ARRAY and 560 were copied into the previous AR. It should be noted that NREF=2 for the array referenced by 560.

3- The instruction counter was changed to 35.

BIBLIOGRAPHY

Gries, David Compiler Construction for Digital Computers (New York: John Wiley & Sons, Inc.,1971),P.173,195.

Meyer,James R.,'Walk-Through of Translator Algorithms.' Unpublished Master's report,Kansas State University,1977.

APPENDIX A

Design Specification Language

Structured programming techniques were used to write the formal algorithms shown in appendix B. It became necessary to add assertions to the algorithms in an attempt to define what was known about the charteristics of the stack and the allowable TAGS. The term ASSERT is used to state a condition which must be TRUE at that point in the program. TRESSA is simply ASSERT spelled **backwards** and therefore makes a logical closure.

CASE statements are used frequently to facilitate future changes to the Algorithms. IF THEN ELSE statements are used ,and the standard BEGIN END is used as a control feature.

The following notation applies to the stack repesentations described in the body of the report:

NOTATION	**MEANING**
AR	**The logical address of the current AR.**
LAR	**The logical address of the last AR. In general, it is used as a pointer to the previous or calling AR. Sometimes it points to the current AR.**

NOTATION	MEANING
AR.subfield	The contents of the specified subfield in the current AR header. Example: AR.ARINST refers to the content of the instruction subfield.
AR.INDEX.subfield	The contents of the symbol table entry identified by the relative index and the specified subfield. Example: AR.INDEX.TYPE refers to the content of the referenced variable's TYPE subfield.

The following notation is used to simplify writing the high level algorithms:

NOTATION	MEANING
OP⟨f1,f2,-⟩	f1 refers to the first subfield of the operand, etc.
OP ⟨f11,f12,-⟩ ⟨f21,f22,-⟩	f11 refers to Operand1 subfield1, f22 refers to Operand2 subfield2,etc, where the first integer identifies the operand and the second integer identifies the subfield.
f21⇒ VAR|SCALAR|TEMP1	Operand2 subfield1 must be variable,scalar, or a temporary.
'TEMP1= VAR| SCALAR'	The quotes ,', are used for comments. This sample comment states that the TEMP (top of the execution stack) must be a variable or scalar.
PUSH⟨TYPE,(AR.f12.VAL)OP(f22)⟩	This is actually two push operations: The TYPE is pushed; The result of some operation is pushed (VAL subfield of the variable specified by the second subfield of Operand1 operating on the second subfield of Operand2).

NOTATION	MEANING
INST_LEN	Word length of the instruction in the particular implementation.
PRCTAB	Refers to the procedure table.
PINDEX	Relative index of the procedure in the procedure table.
NINDEX or INDEX	Relative index of the variable in the activation record.
NEW_ADDR	The new starting address for an array which has been copied.
OBJECT(NEW_ADDR)	Refers to the array which begins at the specified address. The array header can be examined to determine characteristics of the array.

MATRIX NOTATION

The first integer identifies f11's type and the second integer identifies f21's type where; 1=VAR, 2=SCAIAR, and 3=ARRAY. The following matrix sumarizes the notation:

		f21		
		VAR	SCALAR	ARRAY
f11	VAR	1,1	1,2	1,3
	SCALAR	2,1	2,2	2,3
	ARRAY	3,1	3,2	3,3

Example: (1,2) means f11 is a variable and f21 is a scalar.

APPENDIX B

FIGURE 1. Formal Algorithms for the Assignment Instruction

General form: ASGN<f11,f12,-><f21,f22,->

```
'1' PROCEDURE ASSIGNMENT
'2' ASSERT f11=>VAR
       '3' f12=>INDEX
       '4' f21=>VAR|SCALAR|TEMP1   'TEMP1=>VAR|SCALAR'
       '5' f22=>INDEX|TEMP2        'TEMP2=>INDEX|VAL'
'6' TRESSA
'7' IF AR.f12.TYPE=ARRAY
       '8' THEN CALL DEALLOCATE(AR.f12.VAL)
'9' CASE
    '10'(f21=ARRAY)
       '11' BEGIN
          '12' CALL COPY_ARRAY(AR.f22.VAL,NEW_ADDR)
          '13' AR.f12.TYPE <= ARRAY
          '14' AR.f12.VAL<=NEW_ADDR
       '15' END

    '16'(f21=VAR)
       '17' BEGIN
          '18' IF AR.f22.TYPE=ARRAY
             '19' THEN BEGIN
                   '20' CALL COPY_ARRAY(AR.f22.VAL,NEW_ADDR)
                   '21' AR.f12.TYPE<=ARRAY
                   '22' AR.f12.VAL<=NEW_ADDR
                '23' END
             '24' ELSE BEGIN
                   '25' AR.f12.TYPE<=AR.f22.TYPE
                   '26' AR.f12.VAL<=AR.f22.VAL
                '27' END
       '28' END

    '29'(f21=SCALAR)
       '30' BEGIN
          '31' AR.f12.TYPE<=f21
          '32' AR.f12.VAL<=f22
       '33' END
'34' ENDCASE
'35' AR.ARINST<=AR.ABINST+INST_LEN
'36' ENDPROC
```

APPENDIX B

FIGURE 2. Formal Algorithms for the

Monadic Arithmetic Instruction

General form: CF<f1,f2,->

```
PROCEDURE MONADIC_ARITH
ASSERT   F=>VAR|SCALAR|TEMP1     'TEMP1=>VAR|SCALAR'
         f2=>INDEX|VAL           'TEMP2=>INDEX|VAL'
TRESSA
CASE
     (f1=VAR)
         BEGIN
           ASSERT f2=INDEX;TRESSA
                  TYPE can be determined based on
                  AR.f2.TYPE and the operator.
           TRESSA
           PUSH<TYPE,OP(AR.f2.VAL)>
        END
     (f2=SCALOR)
         BEGIN
            ASSERT TYPE can be determined based on f1
                   and the operator.
            TRESSA
            PUSH<TYPE,OP(f2)>
         END
ENDCASE
AR.ARINST<=AR.ARINST +INST_LEN
ENDPROC
```

APPENDIX B

FIGURE 3. Formal Algorithms for the Dyadic Arithmetic Instruction

General form: OP<f11,f12,-><f21,f22,->

```
PROCEDURE DYADIC_ARITH
ASSERT  f11=>VAR|SCALAR|ARRAY|TEMP1
             'TEMP1=>VAR|SCALAR|ARRAY'
        f12=>INDEX|VAL|ADDR|TEMP2
             'TEMP2=>INDEX|VAL|ADDR
        f21=> same as f11
        f22=> same as f12
TRESSA
CASE
     (1,1)
        PUSH<TYPE,(AR.f12.VAL)OP(AR.f22.VAL)>

     (1,2)
        PUSH<TYPE,(AR.f12.VAL)OP(f22)>

     (1,3)
        BEGIN
          CALL COPY_ARRAY(AR.f22.VAL,NEW_ADDR)
          OBJECT(NEW_ADDR)<=(AR.f12.VAL)OP(OBJECT(NEW_ADDR)
          PUSH<TYPE,NEW_ADDR>
        END

     (2,1)
        PUSH<TYPE,(f12)OP(AR.f22.VAL)>
     (2,2)
        PUSH<TYPE,(f12)OP(f22)>

     (2,3)
        same as case(1,3)
     (3,1)
        same as case(1,3)
     (3,2)
        same as case(1,3)
     (3,3)
        same as case(1,3)
ENDCASE
AR.ARINST<=AR.ARINST +inst_LEN
ENDPROC
```

APPENDIX B

FIGURE 4. Formal Algorithms for the Relational Instruction

General form:OP<f11,f12,->><f21,f22,->

```
PROCEDURE RELATIONAL
ASSERT  f11=>VAR|SCALAR|TEMP
             TEMP=>INT|REAL
             SCALOR=>INT|REAL
        f12=>INDEX|VAL|TEMP2
             TEMP2=>INDEX|VAL
        f21=> same as f11
        f22=> same as f12
TRESSA
IF f11=VAR
     THEN BEGIN
            ASSERT f12=INDEX1;TRESSA
            CASE
               (f21=VAR)
                    ASSERT f22=INDEX2;TRESSA
                    PUSH<BOOL,(AR.INDEX1.VAL) OP (AR.INDEX2.VAL)>

               (f21=TEMP|SCALAR)
                    PUSH<BOOL,(f12) OP (f22)>
            ENDCASE
          END
     ELSE BEGIN
            PUSH<BOOL,(f12) OP (f22)>
          END
AR.ARINST<=AR.ABINST + INST_LEN
ENDPROC
```

APPENDIX B

FIGURE 5. Formal Algorithms for the Logical Instruction

```
General form: MCNOP<f11,f12,->
              DYOP<f11,f12,-,><f21,f22,->

PRCCEDURE LOGICAL
ASSERT f11=>VAR|TEMP
            TEMP=>BCOL
       f12=>INDEX|TEMP
            AR.INDEX.TYPE=>BOOL
            AR.INDEX.VAL=>BOOLEAN CCNSTANT
            TEMP=>BCOLEAN CONSTANT
       f21=> same as f11
       f22=> same as f12
TRESSA
IF f21=null
    THEN BEGIN
           CASE
              (f11=VAR)
                  PUSH<BOOL,OP(AR.f12.VAL)>
              (f11=BOOL)
                  PUSH<BOOL,OP(f12)>
           ENDCASE
         END
   ELSE BEGIN
           CASE
              (f11=VAR,F21=VAR)
                  PUSH<BOOL,(AR.f12.val)OP(AR.f22.VAL)>
              (f11=VAR,f21=BOOL)
                  PUSH<BOOL,(AR.f12.VAL)OP(f22)>
              (F11=BCOL,f21=VAR)
                  PUSH<BOOL,(f12)OP(AR.f22.VAL)>
              (f11=BOOL,f21=BOOL)
                  PUSH<BOOL,(f12)OP(f22)>
           ENDCASE
         END
AR.ARINST<=AR.ARINST+INST_LEN
ENDPRCC
```

APPENDIX B

FIGURE 6. Formal Algorithms for the Control Instruction

General form: CP<f1,f2,->

```
PROCEDURE CONTROL
ASSERT OP=>BR|BRT|BRF|BRI
       f1=>VAR|INT
       f2=>INDEX|VALUE
TRESSA
CASE
     (CP=BR)
         BEGIN
           ASSERT f1=INT;TRESSA
           AR.ARINST=f2
         END
     (OP=BRI)
         BEGIN
           ASSERT f1=VAR;f2=INDEX;TRESSA
           AR.ARINST=AR.ARINST.f2.VAL
         END
     (CP=BRT)
         BEGIN
           IF STACK(STKTOP)=TRUE
              THEN BEGIN
                      IF f1=INT
                         THEN AR.ARINST=f2
                         ELSE AR.ARINST=AR.f2.VAL
                   END
              ELSE AR.ARINST=AR.ARINST+INST_LEN
         END
      (OP=BRF)
         BEGIN
            IF STACK(STKTOP)=FALSE
                THEN BEGIN
                        IF f1=INT
                          THEN AR.ARINST=f2
                          ELSE AR.ARINST=AR.f2.VAL;
                     END
                ELSE AR.ARINST=AR.ARINST+INST_LEN
         END
ENDCASE
ENDPROC
```

APPENDIX B

FIGURE 7. Formal Algorithms for the Stack Instruction

```
General form:LINE<f1>
             PUSH<f1,f2,->
             POP
             LOAD<f1>


PROCEDURE LINE
ASSERT f1=LINE_NUM;AR points to current AR;TRESSA
AR.ARLIN<=f1
AR.ARINST<=AR.ARINST+INST_LEN
ENDPROC



PROCEDURE PUSH
ASSERT f1=>VAR|SCALAR|ARRAY|TEMP1  'TEMP1=>VAR|SCALAR|ARRAY|NARG'
       f2=>INDEX|VALUE|ADDR|TEMP2  'TEMP2=>INDEX|VALUE|ADDR'
TRESSA
STKTOP<=STKTOP+1
STACK(STKTOP)<=f1
STKTOP<=STKTOP+1
STACK(STKTOP)<=f2
AR.ARINST<=AR.ARINST+INST_LEN
ENDPROC



PROCEDURE POP
STKTOP<=STKTOP-2
AR.ARINST<=AR.ARINST+INST_LEN
ENDPROC



PROCEDURE LOAD_HEADER
STKTOP=STKTOP+1
STACK(STKTOP)<=f1
ENDPROC
```

APPENDIX B

FIGURE 8. Formal Algorithms for the AR Instruction

```
General form:PUSHAR<f1,f2,->
             POPAR


PROCEDURE PUSHAR
ASSERT    LAR points to calling AR    '0 if MAIN'
          STKTOP points to the number of arguments
TRESSA
AR<=STKTOP+1
LOAD<AR>                          'Initializes AR.ARTAG'
LOAD<LEN_OF_of_AR_HEADER>         'Initializes AR.ARLEN'
LOAD<0>                           'Initializes AR.ARLIN'
LOAD<len_of_CODE_HEADER+1>        'Initializes AR.ARINST'
PINDEX<=LAR.f2.VALUE
LOAD<PRCTAB.PINDEX.PRCCOD>        'Initializes AR.ARLCCD'
LOAD<PINDEX>                      'Initializes AR.ARPIND'
LOAD<LAR>                         'Initializes AR.ARRET'
LAR<=AR
ASSERT    LAR points to current AR
          STKTOP=LAR+LEN_OF_AR_HEADER
TRESSA
AR.ARLEN<=AR.ARLEN+LEN_SYMTAB-LEN_SYMTAB_HEADER
CALL COPY_SYM(AR.f2.VAL,STKTOP)
ASSERT    STKTOP=LAR+AR.ARLEN;TRESSA
AR.ARINST<=AR.ARINST+INST_LEN
ENDPROC
```

```
PROCEDURE POPAR
IF AR.ARRET=NULL
     THEN DONE
     ELSE BEGIN
            ASSERT  LAR=AR=POINTS to current AR
                    Each element of the argument linkage
                    consist of two one word subfields
            IRESSA
            CALL SEARCH(AR,AR.ARLEN,STKTOP)
            STKTOP<=LAR-2*(NARG+1)-1
            LAR<=AR.ARRET

            AR<=LAR
            ASSERT  AR=LAR=POINT to the calling AR
            TRESSA
            AR.ARINST=AR.ARINST+INST_LEN
          END
ENDPROC
```

APPENDIX B

FIGURE 9. Formal Algorithms for the LINK Instruction

```
General form: FLINK<f1,f2,->
              BLINK<f1,f2,->


PRCCEDURE FLINK
ASSERT     AR points to the called AR
           AR.ARRET points to the calling AR
           COUNT=>Number of arguments previously linked
           TAG1=>relative index to the type field of an
                 element in the argument linkage.
            VAL1=>relative index to the value field of an
                 element in the argument linkage.
TRESSA
TAG1<=AR -2*(NARG+1-COUNT)
VAL1<=TAG1 + 1
LAR=AR.ARRET
IF STACK(TAG1)=VAR
     THEN BEGIN
            ASSERT   STACK(VAL1)=INDEX;TRESSA
            AR.f2.TYPE<=AR.INDEX.TYPE
            IF LAR.INDEX.TYPE=ARRAY
                 THEN BEGIN
                        CALL COPY_ARRAY( LAR.INDEX.VAL,NEW_ADDR)
                        AR.f2.VAL<= NEW_ADDR
                      END
                 ELSE AR.f2.VAL<=AR.INDEX.VAL
          END
     ELSE BEGIN
            AR.f2.TYPE<=STACK(TAG1)
            AR.f2.VAL <=STACK(VAL1)
          END
LAR=AR
ENDPROC
```

```
PROCEDURE BLINK
ASSERT     f2 - 1  identifies the placement of the argument
                   in the linkage

           AR.ARRET points to the previous AR, LAR
               AR   points to the current AR
           POPAR    deallocates all address
                    references in and below the
                    current AR.
TRESSA
TAG1<=AR-2*(NARG+1-f2)- 3
VAL1 <=TAG1 + 1
ASSERT      STACK(VAL1)=INDEX                'In previcus AR'
TRESSA
INDEX<=STACK(VAI1)
LAR<=AR.ARRET
IF LAR.INDEX.TYPE=ARRAY
     THEN CALL DEALLOCATE(LAR.INDEX.VAL)
     ELSE NULL
IF AR.f2.TYPE=ARRAY
     THEN BEGIN
            LAR.INDEX.TYPE=AR.f2.TYPE
            LAR.INDEX.VAL=AR.f2.VAL
            CALL DEALLOCATE(AR.f2.VAL)
          END
     ELSE BEGIN
            LAR.INDEX.TYPE<=AR.f2.TYPE
            LAR.INDEX.VAL<=AR.f2.VAL
          END
ENDPROC
```

EXECUTION MODELS FOR TRANSLATOR DESIGN

BY

MILES T. CLEMENTS JR.

B.S., NORTH GEORGIA COLLEGE, 1965

AN ABSTRACT OF A MASTER'S REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

KANSAS STATE UNIVERSITY

MANHATTAN, KANSAS

1977

ABSTRACT

This report presents part of the instruction set for a virtual machine, called CS700, which has been used as the target machine for an interactive interpreter in the CS700 class, Translator Design I. The report contains discussion of the machine architecture, discussion of selected instructions, walk-thrus of the execution of the instructions, and (in the appendix) formal algorithms for the instructions. Selected source lines and the resulting generated code are used as a framework for the execution.

Instructions included are assignment, monadic and diadic arithmetic operators, branching instructions, activation record initation and completion, and linking of arguments between invoked modules. Indexing and I/O instructions are not included.