RICE UNIVERSITY

CODE GENERATION FOR NEW ON THE
RICE RESEARCH COMPUTER

by

Michael Kelly Donegan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

Thesis Director's signature:

Edward Alvin Feustel

Houston, Texas

May, 1972
ABSTRACT

CODE GENERATION FOR NEW ON THE RICE RESEARCH COMPUTER

by

Michael K. Donegan

A code generation scheme is given for the implementation language NEW to be used on the Rice Research Computer. A discussion of the run-time organization of NEW programs is given which provides for independent compilation of routines and generalized jumps using label variables. The code generator is based on an algorithm given by Sethi and Ullman, which has been extended to aid in efficient compilation of NEW constructs. Modes of elaboration are introduced to aid in the handling of assignments and jumps. A heuristic scheme is used for register allocation based on context. Finally, some problems with the language and the computer are discussed and areas for future work are presented.
ACKNOWLEDGEMENTS

The author would like to thank Scott Warren for many helpful hours of discussion and criticism. I would also like to thank Dr. Edward Feustel for his encouragement and help, and to Dr. Kenneth Kennedy and Dr. R. C. Minnich for serving on my Examining Committee.

Finally, I would like to thank my wife, Jackie, for her love and patience.
TABLE OF CONTENTS

Introduction

I. Values in NEW and their Relationship to the R2
   1.1 Values and their representation
   1.2 The Typeless Language in a Tagged Environment
   1.3 VAL and REF Mode in NEW

II. Scope Rules in NEW and Run-time Organization of Storage
   2.1 Introduction
   2.2 REF-VAL Relationships
   2.3 Identification
   2.4 Conditional Identification
   2.5 Parameters in NEW
   2.6 Vector Values in NEW

III. Organization of the NEW Object Program
   3.1 Design Considerations
   3.2 Intersegment Communication
   3.3 Segment Organization
   3.4 Summary

IV. An Introduction to the Code Generator
   4.1 Construct Classes
   4.2 The Syntax Tree
   4.3 The Symbol Table
   4.4 Construct Generators
   4.5 More Elaboration Modes
4.6 An Optimization

V. The Sethi and Ullman Algorithm for Optimal Expression Evaluation

5.1 Motivation

5.2 Basic Assumptions

5.3 The Initial Algorithm

5.4 Commutative Operators

5.5 Operators Which Are Both Commutative and Associative

5.6 Fitting the Model to the R2

5.7 Extensions to Other Nodes

5.8 The GETWGT Routine

VI. Labels and Jumps in NEW

6.1 Introduction

6.2 Intrasegment Jumps

6.3 Label Variables and Intersegment Jumps

VII. The TRANS Routine

Summary and Conclusion

APPENDICES

A. A Collected Syntax for NEW

B. The R2 Instruction Set

C. Tree Nodes in NEW
In this thesis, a code generation scheme is presented which is used to compile the NEW language for the Rice Research Computer (or R2). Both the language and the computer are the result of research done at the Laboratory for Computer Science and Engineering. NEW is an "implementation" language for the R2, intended to be used primarily for writing operating systems and compilers for the new machine. The NEW compiler and its associated run-time system are written in the NEW language.

According to Gries [1], "the hardest and 'dirtiest' parts [of a compiler] are the semantic analysis, program preparation and code generation." One might assume that this is largely due to the "machine independence" of most common languages. For example, one would have great difficulty in implementing a language with character-handling facilities on a machine without character hardware. However, there are more subtle difficulties. Consider, for example, the operator + which appears in most languages to represent addition. Every computer has an ADD instruction; in fact, most large computers have at least two such instructions, one for integer and one for floating point addition. Depending on the number of numeric representa-
tions supported by the machine hardware, there might be several of these. The IBM System/360 has fourteen ADD instructions. The point is that hardware design has as large an effect on the code generation task as the language design.

The R2 is particularly well-suited for high-level language implementation. To use the previous example, there is just one ADD instruction, which may be applied to integral, real, double precision and complex operands. This is possible because data words are self-identifying in memory. A hardware tag, which is automatically interpreted by the machine, can identify a datum as being an integer, an address, a label, and so on. In the case of arithmetic instructions, conversion from integral to real is automatic when required. Double length operands (double precision and complex), when used, cause an automatic trap to software routines for handling. The instruction set itself includes a powerful complement of arithmetic and boolean operations. The use of address words to describe arrays and a variety of data manipulation instructions provide convenient data transfer and powerful operand fetching and storing capability. Addressing operations provide controlled manipulation of addresses with automatic bounds checking to prevent erroneous or malicious data referencing. The branching instructions allow testing of various machine conditions and provide simple subprogram entry and exit.
A hardware stacking mechanism in the machine provides a convenient means for dynamic storage allocation. In addition, recursive routines are easily supported along with nested interrupt handling. Multi-programming is possible on the R2, with each program being provided with a global process base. This global base may be used for intersegment communication. The hardware uses this base for locating trap routines in the event of program errors. Therefore, each programmer may provide his own error handling routines if he so desires.

Although the R2 has many desirable features, it would not be realistic to give the impression that all is perfect. In designing the code generator for NEW, several inadequacies have been discovered in the machine which will be discussed in a later section. However, the overall design of the machine has proved to be quite conducive to efficient high-level language implementation.

One of the primary goals in the design of the NEW language was to make implementation straightforward and efficient on the target machine. Therefore, many of the constructs and facilities in the language are machine dependent. The programmer has been provided access to virtually every feature of the machine. In this sense, NEW is something of a high-level assembler for the R2. The NEW programmer is expected to write efficient algorithms, making use of the features of the R2. Therefore, optimization in the NEW compiler is, at present, restricted to questions of
local efficiency, that is, choosing optimal code sequences for particular constructs, elimination of compiler generated redundancies and inefficiencies, and allocation of registers based on local context information. This thesis deals with how this has been achieved in the NEW compiler. It is believed that the methods used should be useful in the development of compilers for other languages as well.

Much of the material in this thesis, particularly the run-time organization of NEW programs, has developed from countless hours of discussion with Scott Warren, the principal designer of NEW. In addition, many optimization techniques, in particular, the handling of cascading jumps, have resulted in this manner. Since the language is highly machine dependent, the design of the language and of the code generator have been inextricably meshed. The use of and extensions to the Sethi and Ullman algorithm \[3\], the introduction of artificial elaboration modes, and the priority scheme for register allocation, however, are principally the work of this author.

It is assumed that the reader has some knowledge of the NEW language and of the R2. For information regarding NEW, the reader is referred to the manual [4]. The R2 is based on the Basic Language Machine, which is described in Iliffe [5]. A more detailed description of the machine may be found in the preliminary manual [6], while some programming examples and explication of the various machine features may be found in Feustel [7]. For conven-
ience, a collected syntax for NEW and a list of the R2 instructions may be found in the Appendices.
"What kind of code are you going to turn out for this?"

--Scott Kipling Warren
CHAPTER I
VALUES IN NEW AND THEIR RELATIONSHIP TO THE R2

1.1 VALUES AND THEIR REPRESENTATION

All values in NEW are represented in the machine and manipulated at run-time as single word items. These values are divided into four classes, matching those which are distinguished by the hardware. They are: numeric, address, control, and undefined.

Numeric values are those which are acceptable to the arithmetic processor of the R2. Included in this class are integral, real, boolean, double precision and complex. The last two, double precision and complex, are included in this class for convenience. Operations on these values are not directly performed by the hardware, but may be handled by software routines. These routines are called via an arithmetic interpretation trap which supplies them with the operands and the operation to be performed. Of course, these routines may be written in NEW.

Address values refer to vectors of contiguous words in the R2 memory. All operands which reside in the memory must be accessible via an address word, since only limited access to actual memory locations is provided. The values to which an address word refers are available by indexing through that address word. For this purpose, the hardware provides several indexing instructions. Sounds
checking is provided in the hardware to prevent one from accessing any values other than those which properly belong to him. In addition, an initial index field is provided in the address word, thus providing the programmer the ability to have arrays with lower bounds other than zero.

Control values are used to point to entry points in routines and labelled instructions. All jumps from one independent segment to another must find their destination via a control word. Control values also provide the means for subroutine linkage and procedure calls. Whenever a jump is made to a routine, the R2 generates a control word which retains the necessary information to return to the point of the call and restore the machine to its previous control state.

Undefined values are those which may not be used for normal interpretation. Label values and routine values in NEW are hidden by tagging them undefined. This renders them useless to the non-privileged user. If their values are needed, they are properly tagged by a special GOTO-SOLVER routine which then carries out the necessary jump. In this manner, the NEW system protects itself from invalid control, since the GOTO-SOLVER routine may check for a valid environment and restore it properly.

1.2 THE TYPELESS LANGUAGE IN A TAGGED ENVIRONMENT

NEW is a typeless language, that is, the programmer never explicitly identifies his program variables as being of a certain class. There is no permanent relationship
between a variable name and the type of value that it represents. Since the NEW compiler can only assume that the values used will be of the proper type at run-time and turn out object code accordingly, one must rely on error checking at run-time. This would not be feasible using the classical von Neumann architecture. However, the R2 with its tagged architecture provides self-identification of words in storage and provides a good environment for such a language. The R2 hardware performs extensive error checking on all operations. In the case of arithmetic operations, it provides automatic conversion from integral to real, preventing mixed mode errors. All is not perfect, however, since some instructions may inhibit the error checking mechanisms. These operations must be excluded or checked for validity at run-time if non-privileged users are to be allowed to write NEW programs. Since these operations are of little value to the normal user, there should be no harm done by eliminating them.

1.3 VAL AND REF MODE IN NEW

An address word in the R2 which holds the address of another word in memory may be thought of as representing two values, its value as an address word, and the value of the word to which it refers. All variables in NEW may be thought of as representing a pair of words, its value, and a reference to that value. This will be represented diagrammatically as
Here, the diamond shape represents an address word or a reference to the value of A. The square shape represents the value of A, which is not an address here, but may be, as later examples will show. Whenever a variable appears in a NEW source program, it is necessary for the compiler to distinguish between the two values and determine which is needed. Unlike some languages, notably 3LISS [8], the programmer need not specify explicitly which is required in the common situations. However, he may indicate deviations from the normal interpretation if desired. In order to determine the correct interpretation, the compiler makes use of elaboration modes, which are determined by context. Each syntactic element in NEW has associated with it an elaboration mode which governs how it is interpreted by the compiler. The primitive elaboration modes are REF mode and VAL mode. A variable which is elaborated in REF, or reference, mode yields an address word which refers to its value. Similarly VAL, or value, mode elaboration yields the value itself. The simplest example of the use of these elaboration modes is the assignation, of which A := B is an example. The syntax for an assignation in NEW is
VARIABLE assignation: reference destination, becomes symbol, value source.

The semantic interpretation of the assignation is embodied in the syntax by asserting that the destination is to be elaborated in reference mode and the source in value mode. VARIABLE is a metasyntactic variable whose rule is

\text{VARIABLE: reference; value.}

This asserts, in the case of assignations, that the value of an assignation may have two interpretations. The value in reference mode is the value of the destination, while in value mode it is the value of the source. Using the diagrammatic scheme introduced above, the assignation \( A := B \) is represented as

\[\begin{array}{c}
A' \\
\downarrow \\
A
\end{array} \quad := \quad \begin{array}{c}
B' \\
\downarrow \\
B
\end{array}\]

Notice that \( A \), being a reference destination, finds its value at \( A' \) (represented by the arrow from \( A \)), while \( B \), being a value source, finds its value at \( B \). The assignation proceeds from the source to the location referenced by the destination (represented by the dotted line).

Since the value of a new variable may be an address
word, one would like the ability to provide for added indi-
rection or to prevent the indirection which is implied by
VAL mode elaboration. This is done in NEW by the use of
two special monadic operators: REF and VAL. The REF
operator causes its operand to be elaborated in reference
mode. Notice that REF is idempotent, that is, REF REF A
is equivalent to REF A. The VAL operator provides for
extra indirection. Its operand is first elaborated in
the mode dictated by the context (the result must be an
address word). Then the value referred to by the resulting
address word is taken as the value of the formula. Notice
that VAL A, when elaborated in reference mode, is identical
to A elaborated in value mode. An example of the use of
these operators should clarify their meanings. Consider
the NEW program

(LOCAL a, b, c:=2;
    a:=REF c       # the value of a is now a 'pointer' to
c               #;
    b:=VAL a       # the value of b is set to 2 #;
    VAL a:=3       # a's value becomes 3 #;
    a:=c:=2        # a and c now have the value 2 #;
    a:=REF (c:=0)  # c is set to 0 and a again has
                    # a pointer to c #).

Figure 1 illustrates the elaboration of this program in the
diagrammatic form. The last two examples illustrate the
difference between an assignation in VAL mode and one in
REF mode. The assignation c:=2 is elaborated in VAL mode
according to the syntax and its value is 2. The assignation
c := 0, on the other hand, is evaluated in HEP mode, because of the presence of the REF operator, and its value is c'.

The understanding of these two modes of elaboration is fundamental to the understanding of NEW programs. Much of what follows is concerned with the manner in which these elaboration modes affect the meaning of various constructs in NEW.
Figure 1. VAL and REF mode elaboration.
2.1 INTRODUCTION

In a block structured language, such as NEW, storage is taken for values when declarations are encountered. A declaration has the dual purpose of reserving space and establishing a relationship between the tag, or identifier, of the declaration and the location of the new value. This relationship was illustrated by the diagrams in the previous chapter. The area of the program where this relationship is valid is called the scope of the declaration. In this chapter, the scope of declarations in NEW will be discussed, and the means for implementing the scope rules in the machine will be shown.

2.2 REF-VAL RELATIONSHIPS

As was seen in the discussion of REF and VAL mode elaboration, there are actually two relationships to consider. The first is that between the tag and the REF-VAL pair; the second is the pointing relationship between a reference and its corresponding value. This second relationship must be considered since the value of a variable in NEW may be a reference to another value. Hence, one must consider the following questions:

1. What is the duration of the relationship between a tag and the REF-VAL pair it represents?
2. What is to be done about pointers which refer to values which are no longer defined?

3. How may we insure that these pointers will not destroy the integrity of the system?

2.3 IDENTIFICATION

In order to answer these questions, some terminology for discussing scope must be developed. For this purpose, the terminology used by van Wijngaarden [9] in describing the semantics of ALGOL 68 will be adopted. The occurrence of a tag in a NEW program is either a defining occurrence or an applied occurrence. A defining occurrence may occur in NEW either explicitly, as in a declaration, or implicitly, as a formal parameter or in an indexer. All other occurrences are applied occurrences. The scope of a declaration in NEW is defined in terms of ranges. A range is either a serial clause, a routine denotation, or a repetition. The reach of a particular range is that range with the exclusion of all ranges properly contained in it. The scope of a defining occurrence is the smallest range in which it occurs.

Given an applied occurrence of a tag, it must be possible to locate the REF-VAL pair which it represents. This is done by finding the defining occurrence which identifies that tag in the following steps:

Step 1: The applied occurrence of the tag is called the home, and Step 2 is taken.

Step 2: If there is a smallest range which properly contains the home (i.e. if the home is not the program), then that range is considered;
its reach is called the home and Step 3 is taken. Otherwise, the applied occurrence has no defining occurrence.

Step 3: If the home contains a defining occurrence of the given tag, then this is the defining occurrence of the given applied occurrence and the declaration containing the defining occurrence is said to be the declaration of the applied occurrence. The defining occurrence is said to identify the applied occurrence and identification is complete. Otherwise, Step 2 is taken.

In order for this procedure to work, a context condition is placed on NEW programs. This condition is that no reach may contain two defining occurrences of the same tag. This means that (LOCAL a; STATIC a; ...) is improper. The procedure for identification answers the first question posed above. For example, consider the program

(LOCAL x:=3;
  (LOCAL a; a:=x);
  x:=a).

This program is improper because the last occurrence of a has no defining occurrence. However, in the inner range, both x and a have defining occurrences and the assignation a:=x is proper.
2.4 CONDITIONAL IDENTIFICATION

In NEW programs, identification is only a necessary condition for an applied occurrence to be proper. It is insufficient because up-level addressing is partially prohibited in the NEW language. In NEW, there are two storage classes, local and global. The NEW programmer declares a variable to be in one of these classes by choosing the proper declarer. Variables declared with LOCAL and all implicit declarations belong to the local class, while STATIC and EXTERNAL declarers are used to declare global variables. These classes are implemented differently by the compiler. Global variables reside on the global base and the space declared there is permanent or static. Local variables, on the other hand, are handled dynamically by being placed on the hardware stack or in registers. This stack is also used by the hardware to store return links whenever a routine is called. When this happens, the stack length is automatically set to zero, introducing a new stack regime. Thus any locals which were accessible before the call are now inaccessible. This is unfortunate, but rather than provide a more complicated mechanism, it was decided not to permit access to locals declared outside a routine from within that routine. This requires that the procedure for identification be expanded as follows:

If an applied occurrence identifies a defining occurrence (in the sense defined above) and the
declaration of the applied occurrence is not a STATIC or EXTERNAL declaration, then the applied occurrence is conditionally identified by that defining occurrence. If an applied occurrence which occurs in a routine denotation is conditionally identified by a defining occurrence not contained in that routine denotation, then the applied occurrence has no defining occurrence.

According to the second procedure, global variables in a routine may identify declarations which occur outside the routine. This is true because global locations are always assigned to static locations in memory. These locations are never allowed to conflict and are reserved until the program is terminated. For this reason, pointers to these locations are always valid. Of course, the tags associated with global declarations have the same scope rules as tags for local declarations. To resolve the second question, pointers to local values must be considered. For example, consider the program

(LOCAL mm;
  (LOCAL a; mm:=REF a);
  (LOCAL b; VAL mm:=3)).

In the second line, a pointer to a is assigned to the variable mm. Immediately following that, the space reserved by a is relinquished because the end of its scope-defining range is reached. If a strict stack allocation scheme is
followed, \( b \) will be given the space that was previously assigned to \( a \). Thus the last assignation will result in \( b \) being set to the value \( 3 \). This effect is certainly undesirable. First of all, it is not clear what the program will do, unless one is familiar with the internal workings of the compiler, or has a listing of the object code. That is, the result of the last assignation is not defined in NEW, but only in terms of the model used to run NEW programs. Note that our explication of the result of this program being elaborated hinges on the assumption that one uses a stack discipline for assigning local space. Secondly, even if it were common knowledge that our assumption held true, it would still not be immediately obvious what the program would do. One can easily think of much more devious examples.

For the present, this problem has been ignored in the NEW compiler. That is, the result is said to be undefined. In some instances, the desired effect may be achieved, while in others, it will not. However, it is assured that such undefined references will cause no errors with consequences outside the program in which they occur. The worst that could happen is that the program will possibly cause an invalid indexing error.

2.5 PARAMETERS IN NEW

How may one communicate local information to routines in NEW? Of course, one may do this by assigning the local value to a global location. More commonly, it will be passed as a
parameter. The MEW parameter passing mechanism is designed to be powerful and fast. There are two types of parameters provided, normal and register.

Normal parameters are evaluated in a special mode which is defined in terms of the REF and VAL modes discussed earlier. This mode is called PARAM mode. PARAM mode provides for passing parameters by reference whenever possible. The value of a parameter in PARAM mode is obtained in the following manner:

Step 1: The parameter is elaborated in reference mode, if possible. If the elaboration is improper in reference mode, then Step 2 is taken. Otherwise, the address word obtained is tagged as a chained address word. This chained address word is the value of the parameter in PARAM mode.

Step 2: The parameter is evaluated in value mode and the value obtained is the value of the parameter in PARAM mode.

Normal parameters are evaluated one at a time and placed in a parameter vector and the address word referring to it is passed in register X14. A MEW programmer may have access to this address word by supplying a name for it in his routine denotation. Since there is no check in MEW that the number of formal and actual parameters match, one may write a generalized maximum function as follows:
PROC (;v: # this is the name of the parameter vector since it is followed by a colon #):

(LOCAL m:=16"300000000000" # the smallest integer value in the R2 #;

FORALL x IN v DO # step through the parameter vector #

IF x GT m THEN m:=x FI; M # the result is the value of M#).

Notice that this program will work for any numbers of parameters, including none. When there are no parameters, the value is the smallest integer possible. Also notice that none of the parameters are named; all are referred to indirectly by the use of the parameter vector option.

Register parameters are always passed by value. The reason for this is that when a chained address word is placed in a register, undesirable effects are possible. Of course, the privileged programmer may pass a register parameter by reference if desired as in

F ( REF a HTAG 15).

Here a is passed by reference by first evaluating a in REF mode and then setting the hardware tag of the result to 15, which is the tag value of a chained address word. Routines which use register parameters will usually run faster since register access is faster than memory access. However, one must remember that when a register parameter is used, expression evaluation may be hampered.
2.6 VECTOR VALUES IN NEW

Arrays are provided in NEW by use of local and global generators, denoted by VEC and LOCVEC. Since, initially, there will be no operating system on the R2, no provision is made for garbage collecting global vectors. Thus global generators must be used with care. The programmer, however, may elect to take space and manage it himself.

The LOCVEC intrinsic presupposes that X13 initially describes a linear memory segment called the local vector region. When a program is initiated, the programmer may supply the size of this region. When a local generator is used, the compiler generates in-line code to take space. When the range of the generator is exited, code is generated to free that space. Register X13 always refers to the free part of the local vector region. The programmer is wise to recall that a repetitive statement is a range. Thus, if one writes

\[
\text{(LOCAL } v := \text{LOCVEC } (n); \\
\text{ FOR } i \text{ TO } n \text{ DO } v[i] := \text{LOCVEC } (i) \ldots )
\]

expecting to create a triangular array in the local vector region, he may be surprised at the result. The problem is that the space taken in the repetition is deallocated as soon as it is taken. This should properly be written as

\[
\text{(LOCAL } v := \text{LOCVEC } ((n+3) \ast n/2); \\
\text{(LOCAL } p := v \mod n; v \limb n; \\
\text{ FOR } i \text{ TO } n \text{ DO } \\
\text{ (} v[i] := p \limb i; p \modb 1) )\).
\]
CHAPTER III
ORGANIZATION OF THE NEW OBJECT PROGRAM

3.1 DESIGN CONSIDERATIONS

The previous discussion has been concerned with the organization and storage of values in NEW programs. In this section, the organization of the NEW object program itself will be discussed. Among the design considerations that have an effect on this organization are the following:

1) Independent compilation -- It is possible in NEW to compile routines separately and link them together to create a complete program. This makes it possible to change a routine without recompiling the entire program.

2) Re-entrant object code -- Every NEW routine is re-entrant; this allows routines to be shared among any number of processes. Likely candidates for sharing are math library routines and I/O routines.

3) Relocation -- Each NEW routine is relocatable to any location in memory; this allows the operating system to move program segments at will.

4) Logical addressing of program segments -- Routines communicate in NEW via logical segment numbers. The actual address of each routine is kept in a segment table in memory. Therefore, moving a program segment only involves updating one pointer.
In addition, segments need not be in memory, but may be read in from secondary storage when needed.

3.2 INTERSEGMENT COMMUNICATION

As mentioned earlier, routines in NEW have limited scope. They may only use values which are parameters, global locations, or their own locals. For this reason, routines have been chosen to be the basic NEW object segment. An object module, which is the output of the compiler, consists of a number of these segments and any linkage information.

Separately compiled routines and even routines which occur in the same program may communicate through the global base. This is done via EXTERNAL declarations in the program. The EXTERNAL declaration assigns a location on the global base and gives it an internal and external name. The external name is used in the link editing process, while the internal name is used by the programmer. If no external name is specified, then the internal name is taken as the external name (truncated to 12 characters, if necessary). During the link editing process, the external references are all made to refer to a common location on the global base.

Routines must also be able to call one another. Since each routine is a separate segment, it must be possible to identify the called routine and locate it in memory. For this reason, routines are accessed by a unique segment number. A variable which yields a routine
as its value simply contains the segment number of that routine. In order to locate the routine, a special GOTOSOLVER routine is provided which finds the address of the routine in the segment table. Unfortunately, this must be done for every call, but this is more than offset by the ability to easily relocate program segments. Since all segments have logical names, a single segment or a number of segments may be recompiled separately. The link editor may simply replace a given segment in an object module with the newly compiled segment. Figure 2 illustrates the use of the segment table.

3.3 SEGMENT ORGANIZATION

Since the R2 addressing rule provides relative addressing and address and control words may have relative offsets, segments may be made completely relocatable. All references which are external to the segment must be made via address words which reside in the registers. To make segments re-entrant, the code generator must insure that no value in a segment may be altered. In the case of strings, the value of which is an address word describing the string, the write-protect bit is set on the value to prevent inadvertent changes. Figure 3 illustrates the various parts of a segment and its relation to other regions. As this figure shows, the segment is divided into four parts, the entry table, a segment linkage table, the object code, and the data vector. The entry table contains control
words describing various entry points into the object code. This will be described later when jumps are considered. The segment linkage table holds the constant value of a routine denotation. It is separate from the data vector to make the segment number easily accessible to the link editor.

3.4 SUMMARY.

In the previous sections, the representation of values in the R2 and classes of storage have been discussed. The organization of the MEW programs has been shown to have some useful properties which aid in writing programs by providing independent compilation of routines. In the next section, the object code for MEW programs will be discussed and some local optimizations will be presented.
Figure 2. The Logical Segment Space
Figure 3. Program Segment and Related Areas
CHAPTER IV

AN INTRODUCTION TO THE CODE GENERATOR

4.1 CONSTRUCT CLASSES

Up to now, we have been considering the primitive values in NEW and how they are represented in the machine. We now turn our attention to how these values are to be manipulated. For this purpose, the constructs of NEW are divided into three classes, data accession, computational, and control.

By data accession, we mean any construct which is used solely for the purpose of addressing data. Aside from accessing bases, there are two data accession constructs in NEW, subscription and field selection.

Computational constructs are those involving operators. These are the expressions of ALCOL-like languages. Here the code generator must evaluate the operands for the operator, and generate the code to evaluate the operator.

Control constructs are used to make decisions, synthesize program loops, link to subroutines, and perform jumps to labels. All control constructs have one thing in common, the use of branching. In addition to the code required to perform jumps, there must be additional facilities to preserve the block structuring when a jump takes the program out of a range.

Before considering the various constructs, we will discuss the internal organization of the code generator.
4.2 THE SYNTAX TREE

The input to the NEW code generator is a syntax tree which is built by the parser. The parser also builds a symbol table and passes the necessary information for its reconstruction in the tree. The syntax tree consists of a number of nodes, one for each construct in the program, each having a set of associated operands. For example, the node for a conditional clause is identified as an IF node and has three operands, representing the choice clause and the true and false arms of the clause.

Denotations, except for routine denotations, do not possess their own nodes, but rather are represented in the operands. To distinguish between a node which represents a nested construct and the various denotations, each operand has a KIND field. Bases are also represented in this way, the value of the operand being a pointer to the symbol table.

Nodes in the tree are identified by means of two fields, STYPE and FTYPE. The first field, STYPE, used to segregate the nodes into the various construct classes. In addition, operators are divided into classes depending on whether they are monadic, commutative, non-commutative, and so on. The reason for this is apparent in the next chapter. The FTYPE field is used to distinguish among members of a class.

4.3 THE SYMBOL TABLE

The symbol table for NEW is designed to facilitate the handling of block structuring and scope checking. It consists of two levels, those symbols which are currently de-
fined, and those which are currently superseded by new definitions at a higher lexical level. Symbols which have been pushed down in this fashion are chained together via the PREVDEFS field. Each symbol is uniquely identified by its position in the symbol table, a range number, and the procedure nesting level. All symbols which are declared in the same range are chained together via the CHAIN field. The beginning of this chain is held in a table, indexed by the range number. Whenever a SERIAL CLAUSE node is encountered, all definitions for the serial clause are loaded into the symbol table and linked to the PREVDEF chain. In addition, the current range number is incremented and a new range is created. After the code for that range has been generated, the reverse operation is performed to restore the symbol table to its previous state.

As declarations are encountered, the symbol table entry is updated to reflect the address of that symbol. If the variable is a LOCAL, then a previous pass has noted when it is desirable for that LOCAL to be placed in a register. If a register is available, the LOCAL is assigned to that register. In order to keep track of register contents, a table is kept which reflects the status of each register. If the register is occupied, the table entry reflects the current contents. If temporary values are assigned to registers, they are assigned value numbers. A separate value table reflects the current location of the value. By proper adjustment of these entries, a value
may be moved without losing track of it. This is useful, for instance, when a register must be freed for more imperative use.

4.4 CONSTRUCT GENERATORS

Each node in the tree has a corresponding generator which is responsible for interpreting the various operands and generating the code for that construct. The generators are similar in nature to Miller's macros [10]. His terminology will be used in discussing them.

The state of the code generator is defined to be the location of all values known to it and the current mode of elaboration. For each generator, there is an input state which is determined by the location of the operands of the corresponding node and the current elaboration mode. Unfortunately, the generators require that the operands be in specific locations. For example, the generator for arithmetic operators requires that one operand be in the accumulator and the other be addressable. In addition, only VAL mode is allowed. The restrictions placed on the operand locations by a generator comprise the permitted states. The permitted states are those configurations which are necessary for the generator to work properly. Before using a generator, Miller chooses a permitted state and causes a transition to that state. In the KEW code generator, this work is done in the generator itself. Once a permitted state is reached, the generator produces the proper code and leaves the result in some predetermined place. The
output state of the generator is the location of its result.

The generators will be discussed in more detail in later sections.

4.5 MORE ELABORATION MODES

Three elaboration modes, REP, VAL and PARAM, have been discussed previously. These modes are those defined in the language. In the code generator, there are other artificial elaboration modes which aid in producing efficient code for some constructs.

In the R2, certain instructions cause the condition code register to be set. Depending on the instruction, the CC may be set in three different ways, arithmetically, logically, or via the hardware tags. In conditional clauses, the choice clause is evaluated to determine which alternative clause is to be elaborated. Rather than evaluate the choice clause to obtain a value, we merely wish to obtain the value in the CC in preparation for a subsequent Jump on Condition Code instruction. For this purpose, rather than VAL mode, the code generator uses two artificial modes, BCC (boolean CC) and ACC (arithmetic CC). There is no special mode for the third CC setting, since evaluation in VAL mode will set the CC properly. Elaboration using these artificial modes results in a much shorter code sequence. If the programmer wishes to use the CC value, there are operators provided in the language for this purpose, ACC, BCC, and CCMODE. These operators cause their arguments to be evaluated in the proper mode and return the
value of the CC register.

There are three other artificial modes used exclusively by the code generator. These are ASSIGN, GOTO and VOID mode. ASSIGN mode is used when the left-hand side of an assignation must be evaluated and the value of the right-hand side has been brought to a register. Associated with ASSIGN mode is a value number which refers to the right-hand side. The advantage is that the left-hand side need not be evaluated in reference mode which is costly. GOTO mode is similar in intent. The alternative would be to evaluate the expression yielding the jump destination in value mode. As we will see later, this can be very costly and should be avoided if possible. Using the special GOTO mode, most programmed GOTO's can be elaborated in at most two instructions. VOID mode is used when an action, such as an assignation, is required, but the value of that action is not required. The VOID mode elaboration prevents possible copying of values in these situations.

4.6 AN OPTIMIZATION

Before the generators are discussed further, we will first divert our attention to an optimization technique which is used in the NEW compiler. This technique was designed primarily for efficient elaboration of arithmetic expressions, but the algorithm is also useful for other NEW constructs.
5.1 MOTIVATION

Because of the disparity in access time between the registers in R2 and memory (36:1 in the best case to 110:1 for the slower memory), one can see that it is desirable to keep the number of core accesses to a minimum. In the NEW code generator, we have adopted the use of an algorithm due to Sethi and Ullman. This algorithm is for optimal generation of code for arithmetic expressions. Assuming that there are no trivial relationships among the subexpressions, they have proved that their algorithm minimizes the number of instructions required and the number of memory accesses to evaluate the expression. They consider only the arithmetic operations add, subtract, multiply and divide. Since all expressions in NEW yield a value, it is useful to extend the technique to the whole of NEW to provide optimal code generation. Before doing this, however, the original algorithm will be discussed.

5.2 BASIC ASSUMPTIONS

The machine that Sethi and Ullman consider has a memory and at least one register where arithmetic operations take place. The instructions permitted are
1) (register) LD (storage)
2) (register) STO (storage)
3) (register) AOP (storage)
4) (register) AOP (register)

Instructions of types 1 and 2 are used to transfer between the registers and memory. Type 3 instructions perform some arithmetic operation using the contents of some register as one operand and some storage location as the other operand. The result is left in a register. Finally, type 4 operations take two register contents as operands and leave the result in some register.

The algorithm assumes the expression to be represented as a binary tree in the standard fashion. In speaking of a binary tree, the terminology used by Knuth [11] will be adopted.

5.3 THE INITIAL ALGORITHM

The algorithm for code generation proceeds in two passes. The first pass is used to label each node of the tree with a weight corresponding to the number of registers required to evaluate its subtree. As an example, we will consider the following tree:

```
+   -   e
  /   /   /
a b  c  d
```

which represents the expression \((a \times b - c/d) + e\). How should the labelling be carried out? First consider the subex-
pression \( a \times b \) and the corresponding subtree

\[
\begin{array}{c}
\times \\
a \\
\end{array}
\]

The code for evaluating this expression might be

\[
\begin{align*}
\text{REG1 & LD & a} \\
\text{REG1 & MPY & b} .
\end{align*}
\]

Notice that this computation requires one register. Thus the weight assigned to the \( \times \) node in this subtree should be 1. In this simple case, we will adopt the convention that the weight of the left son will be 1, while that of the right son will be 0. Thus the example subtree with associated weights would look like

\[
\begin{array}{c}
\times \\
1 & a \\
\end{array}
\]

Now consider the more general case

\[
\begin{array}{c}
. \\
? \\
. n \\
\end{array}
\]

Here, the \( . \) represents any binary operation, and the weights of the subtrees are \( m \) and \( n \). The \( ? \) represents the weight to be determined. In order to calculate the undetermined weight two cases are considered. First assume that \( m \neq n \). Without loss of generality, it may be assumed that \( m \) is strictly greater than \( n \). What happens when the tree is evaluated? First, the left subtree is evaluated which requires \( m \) registers. The value is left in some register \( X_m \). Now the right subtree is evaluated, requiring \( n \) registers. Since \( m \) is strictly greater than \( n \), at most \( m \) registers were used in the evaluation. The second result is left in
$X_n$, and the final result is calculated. If $n$ had been greater than $m$, then the right subtree could have been evaluated first and only $n$ registers would be required. Hence, the weight of the tree is $\max(m, n)$.

Now assume that $m=n$. In this case, whichever sub-expression is evaluated first will have to be saved in one of the registers. But since the weights are equal, the evaluation will require $m+1=m+1$ registers.

The above may be summarized in the following rule for weight assignment:

Let $n$ be some node in the tree representing some expression. Then the weight of $n$, $w(n)$, is defined to be

0, if $n$ is a leaf and a right descendant,
1, if $n$ is a leaf and a left descendant.

Otherwise, let $l_1$ and $l_2$ be the weights of the left and right subtrees of $n$, respectively.

Then $w(n)$ is

$l_1+1$, if $l_1 = l_2$
$max(l_1,l_2)$, if $l_1 \neq l_2$.

From the above rule, the weight of any tree may be simply calculated via an endorder traversal of the tree. As was hinted in the discussion, the weights of the nodes in the expression tree should determine a good order of evaluation. In fact, if there is an unlimited number of registers, the algorithm is as follows:
ALGORITHM 1: Let $n$ be the head of some expression tree which is not degenerate (i.e. $n$ is not a leaf). Let $n_1$ and $n_2$ be the left and right descendants of $n$, respectively. Also let $1 = w(n)$, $l_1 = w(n_1)$, and $l_2 = w(n_2)$. Assume that registers $X_1, \ldots, X_n$ are available.

Case 1: If $l_2 = 0$ (notice $l_1$ cannot equal 0), then perform Algorithm 1 on $n_1$, leaving the result in $X_1$. Now using the value in $X_2$ and the value $n_2$ in storage, evaluate $n$.

Case 2: If $l_1 > l_2$, then evaluate the expression with the higher weight using Algorithm 1, leaving the value in $X_1$. Now evaluate the other expression using Algorithm 1 with $X_2, \ldots, X_n$ available. The result will be left in $X_2$. Now use this value and the value in $X_1$ to evaluate $n$.

Case 3: If $l_1 = l_2$, then consider $l_1$ to be the greater weight and proceed as in Case 2.

Notice that the (seemingly arbitrary) choice of giving a leaf which is a right descendant the weight 0 enables the algorithm to work. This is because the operation specified at $n$ might be non-commutative. This choice forces the algorithm to work without special considerations; it is probably the single most important contribution in the Sethi and Ullman paper. Previously, as in the work done by Nakata [12], all leaves were given the same weight. As a result,
no simple algorithmic scheme could be given.

Let us now consider what happens when there are not enough registers available to carry out the entire computation. To Algorithm 1, we add

Case 4: If both $l_1$ and $l_2$ are greater than $N$, then evaluate $n_2$ using Algorithm 1. Store the result in a temporary location. Finally, evaluate $n_1$ and calculate $n$ using the value stored and the value of $n_1$. Otherwise, proceed as above.

Case 4 prevents any problems with non-commutative operations by storing the value of the right subtree. The original algorithm suffices if only one of $l_1$ or $l_2$ is greater than or equal to $N$. This is because the larger valued subtree will be evaluated first, leaving $N-1$ available registers. This is sufficient to evaluate the lower-valued subtree.

5.4 COMMUTATIVE OPERATORS

If an operator is commutative, then the order of evaluation is irrelevant. This can lead to increased efficiency. Consider the tree

```
+--- a
     |   
     v   
    __b____
        |   |
        v   v
     c
```

Using Algorithm 1, the code generated for this expression would be as follows:
XI LD b
XI SUB c
X2 LD a
X2 ADD X1

Obviously, a better method would be
XI LD b
XI SUB c
XI ADD a

The latter scheme saves both a register and an instruction. The reason for this is that \( + \) is a commutative operator.

If the expression given had been expressed as

\[
\begin{align*}
\text{\( + \)} & \\
\text{\( - \)} & \\
\text{\( a \)} & \\
\text{\( b \)} & \\
\text{\( c \)} &
\end{align*}
\]

then Algorithm 1 would have caused the second code sequence to be generated. This motivates the following:

ALGORITHM 2: If \( w(n) \) is greater than 1, and \( n_1 \) is a leaf, and the operation specified at \( n \) is commutative, then interchange \( n_1 \) and \( n_2 \) in the tree. With the tree altered in this fashion, perform Algorithm 1.

In Algorithm 2, interchange takes place only when \( w(n) \) is greater than 1. If \( w(n)=1 \), then \( n_2 \) is a leaf (since its weight must be 0), and interchanging would be useless.

5.5 OPERATORS WHICH ARE BOTH COMMUTATIVE AND ASSOCIATIVE

The last algorithm presented by Sethi and Ullman is concerned with optimizations involving operators which are
both commutative and associative. This algorithm is not presently used by the NEW code generator, but may be added later. An example should suffice to give the flavor of this algorithm. Consider the expression represented by

\[
\begin{array}{c}
+ \\
/ \\
/ \\
+ \\
a \\
b \\
c \\
d
\end{array}
\]

from which the following code is produced

\[
\begin{array}{c}
X1 \quad LD \quad a \\
X1 \quad ADD \quad b \\
X2 \quad LD \quad c \\
X2 \quad SUB \quad d \\
X1 \quad ADD \quad X2
\end{array}
\]

If we take advantage of the commutative and associative properties of the operator \(+\), then the tree may be reordered as follows:

\[
\begin{array}{c}
+ \\
/ \\
\quad + \\
\quad / \\
\quad a \\
\quad b \\
\quad c \\
\quad d
\end{array}
\]

The code generated for this tree would be

\[
\begin{array}{c}
X1 \quad LD \quad c \\
X1 \quad SUB \quad d \\
X1 \quad ADD \quad b \\
X1 \quad ADD \quad a
\end{array}
\]

This is a significant gain in efficiency. Briefly, this reordering is performed by finding groups of operators in
the tree which are associative and commutative, such that the nodes for the operator form a subtree. Then all the descendants of this subtree are ordered by weight and a binary such as the following one is built:

```
    +   \
   /     \
  +------- +
     \      \n   +------- +
      \      \n    +------- +
       \      \n    +------- +
```

The \( n_i \)'s are decreasing in weight as \( i \) increases. One nice feature of this scheme is that it enables the compiler to recognize such situations as \( 2+a+5 \), transforming this to \( a+2+5 \), and finally to \( a+7 \). Of course, this requires that constant leaves be given special consideration in that reordering scheme. It also requires some kind of constant evaluator to perform the compile-time addition.

Presently, it is felt that this last algorithm is more time consuming than its utility would warrant. However, the final decision cannot be made until more NEW programs have been written and their behavior examined.

5.6 FITTING THE MODEL TO THE R2

Unfortunately, the model used by Sethi and Ullman is not correct for the R2. Since the R2 has only one arithmetic accumulator (rather than all registers being accumulators, as Sethi and Ullman have assumed), operations of the form
AOP (storage) and AOP (register) may only be performed if the left hand register is the accumulator, \( U \). This requires a modification of the algorithms previously given.

Before discussing the necessary changes, however, we introduce a new class of operators. An operator is defined to be semi-commutative if it is not commutative and the following holds:

Given the expression tree

\[
\begin{array}{c}
  \text{a} \\
  \text{\textbackslash} \\
  \text{b}
\end{array}
\]

where \( \_ \) is an arbitrary binary operator and \( a \) and \( b \) are leaves, then there exist machine instructions \( OP_1 \) and \( OP_2 \), such that both of the code sequences

\[
\begin{align*}
  X1 & \text{ } \text{LD } a \\
  X1 & \text{ } OP_1 \text{ } b \\
  X1 & \text{ } \text{LD } b \\
  X1 & \text{ } OP_2 \text{ } a
\end{align*}
\]

will evaluate the given expression correctly.

This definition is obviously machine dependent. An example of such an operator for the R2 is the \( _- \) operator, since there are Subtract and Reverse Subtract instructions. When semi-commutative operators exist, they may be treated as commutative simply by changing the operation whenever the operands are interchanged. This results in the following reformulation of Algorithm 2:
ALGORITHM 2': If \( w(n) \) is greater than 1, and \( n_1 \) is a leaf, and the operation specified at \( n \) is commutative or semi-commutative, then interchange \( n_1 \) and \( n_2 \) in the tree. In addition, if the operator was semi-commutative, then change the operation specified accordingly. With the tree altered in this fashion, perform Algorithm 1.

With this modification in mind, we now consider the problem of having a single accumulator. The problem arises in Case 2 of Algorithm 1. Consider \( X_1 \) to be the accumulator. If a value is left in the accumulator, then no calculations may be done until that value is saved. After the second calculation is performed, its value and the previously saved value may be used to perform the final calculation. Case 2 can be reformulated as follows:

Case 2: If \( l_1 \neq l_2 \), then evaluate the expression with the higher weight using Algorithm 1, leaving the result in \( X_1 \). Now copy this value into \( X_2 \) and evaluate the other expression using Algorithm 1 with \( X_1, X_2, \ldots, X_N \) available. The result will be left in \( X_1 \). If the subtree with higher weight was \( n_2 \), then evaluate \( n \) using the values in \( X_1 \) and \( X_2 \). Otherwise, consider the operation specified at \( n \). If this operation is commu-
tative, then we may evaluate \( n \) directly.

If the operation is semi-commutative, then \( n \) is evaluated using the reverse operation.

If the operation is non-commutative, then exchange \( X_1 \) and \( X_2 \) and evaluate \( n \) with the given operation.

This algorithm is slightly more complicated than the original and leads to code which is less efficient than that given by the original algorithm. Fortunately, the R2 has an operation which permits us to perform the exchange of \( X_1 \) and \( X_2 \) directly; otherwise, a temporary location would be required.

5.7 EXTENSIONS TO OTHER NODES

The Sethi and Ullman algorithm works quite well for FORTRAN-like arithmetic expressions. In ARV, however, expressions may be much more general. For example, the formula

\[
A + \text{IF} \ p=0 \ \text{THEN} \ B \ \text{ELSE} \ 6 \ \text{FI}
\]

with corresponding tree

```
        +
       /|
      /  |
     +   IF
      /|
     /  |
    A   B
     /|
    /  |
   =   6
   /|
  /  |
p 0
```

If a consistent method could be found for assigning a weight to the IF node, then the algorithms given above could be used to evaluate the formula. The IF node is just an example, and the following argument can be used for all other control-type nodes.
Notice that operator nodes represent actions to be performed using the values of their subtrees as operands. Thus to evaluate an operator node, each subtree must be evaluated and the value saved before the actual operation can be performed. Control nodes are different in that they merely hold subcomponents of construct and specify an order of elaboration. The nodes themselves are only markers and specify no action on the operands. In the example above, the IP node merely conveys the need to elaborate p=0, and depending on that value, to elaborate either 3 or 6. The weight for this node is just the maximum of the weights of the subtrees. The general case is as follows:

If a node $n$ is a control node with operands $n_1, \ldots, n_k$, evaluate $w(n_1), \ldots, w(n_k)$. If each $w(n_i)=0$ then assign $n$ a weight of 1, otherwise, the weight of $n$ is $\max(w(n_1), \ldots, w(n_k))$.

Using the above rule, the control nodes may be assigned weights and the algorithms utilized as before. The advantages of this should be clear.

Of course, there are still some other nodes to consider: monadic operators, subscriptions, and field selections. The first two may be considered to be special cases of dyadic operators. The latter is replaced by an equivalent expression involving shifts and boolean operators.
5.8 THE GETWGT ROUTINE

In the NEW code generator, a routine called GETWGT is used to assign weights to the tree nodes. In addition, this pass through the tree is used to gather useful information about the tree. Certain questions are often asked by the generators which involve a fair amount of computation. In order to alleviate this, bits are assigned to these questions in the node itself. Such questions are,

1) Is a particular operand a tree pointer?
2) Does a subtree describe an addressable value?
3) Is it necessary to use the accumulator to evaluate this subtree?

In addition, the GETWGT routine aids in the allocation of locals to registers. In the NEW code generator, locals are assigned to registers whenever possible. However, since the number of registers is limited, a measure of the importance of each local is necessary. The GETWGT routine assigns priorities to locals depending on the context of their applied uses. This is done merely on a heuristic basis and the scheme may require some "tuning" later to achieve the best results. The priority assignments is done first on the basis of the declaration context as follows:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest</td>
<td>Local declarations</td>
</tr>
<tr>
<td></td>
<td>For Indexers</td>
</tr>
<tr>
<td>Highest</td>
<td>Forall Indexers</td>
</tr>
</tbody>
</table>

Indexers have higher priority since they are used in loops.
For all indexers, being address words which must be brought to registers whenever they are used, have the highest priority. Since the use of locals has some bearing on their being allocated to registers, applied occurrences cause the original priorities to change. Each time an applied occurrence is reached, GETWGT increments the original priority by a certain amount, depending on the context as follows:

<table>
<thead>
<tr>
<th>Priority Increment</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest</td>
<td>Simple</td>
</tr>
<tr>
<td></td>
<td>Within a loop</td>
</tr>
<tr>
<td></td>
<td>Primary of a subscription</td>
</tr>
<tr>
<td>Highest</td>
<td>Primary of a subscription</td>
</tr>
<tr>
<td></td>
<td>within a loop</td>
</tr>
</tbody>
</table>

Using the resulting priority, the code generator can then assign locals to registers from the highest priority down, until all registers are allocated.
CHAPTER VI
LABELS AND JUMPS IN NEW

6.1 INTRODUCTION

Labels are available as values in NEW and like other values they may be assigned to variables, stored in arrays, passed as parameters, and so on. Labels are a rather special kind of value, however. Not only must they contain the location of the labelled instruction, but they must also contain information about the environment of that instruction. This is necessary to preserve the block structured nature of the language. There are two types of labels, those which the compiler generates for control constructs, and those which the programmer explicitly declares. There are also two types of jumps, intrasegment and intersegment. This chapter deals with the manner in which labels and jumps are handled by the code generator and with how these are realized in the R2.

6.2 INTRASEGMENT JUMPS

In a block-structured language, jumps present a problem since they may lead the execution out of the current range. If the destination of a jump is contained within the same routine as the jump itself, it is called an intrasegment jump. To perform an intrasegment jump in NEW, all that is necessary is to remove any locals and temporaries which may be on the stack (if the jump is to a new range), and execute a jump to the label. This may be done in at most
two instructions in the R2. In NEW, labels have the same scope as local variables. Hence, jumping into ranges is not allowed.

Label creation and jumping to compiler-generated labels is handled in the NEW compiler by three routines, COMPLAB, JUMPCOND, and NULABEL. NULABEL is merely a unique label generator which returns a number representing the next label. When an instruction is to be labelled, a call on COMPLAB is used, passing the label number as a parameter. In order to generate a jump to that label JUMPCOND is used. The parameters to JUMPCOND are the label number, a condition to test, and a boolean representing whether to jump if the condition is true or false. If no condition is passed, the actual CC value to jump on is passed instead.

There are two problems to be solved with this scheme. First of all, it must be possible to jump forward to a label which has not yet been placed by COMPLAB. To facilitate this, a table of labels is kept which contains the status of each label, either defined or undefined. If a jump to an undefined label occurs, a pointer to the jump instruction is placed in this table. If more than one such jump occurs, they are chained together using the next field. When the label is finally placed, the chain is followed to find the jump instructions and patch them. At the same time, a toggle is set to indicate that the next emitted instruction is labelled. The reason for
this is to prevent a condition called cascading jumps. To illustrate the problem, consider the conditional clause

\[
\text{IF } a=0 \text{ THEN IF } b=0 \text{ THEN } c \text{ ELSE } d \text{ FI ELSE e FI .}
\]

In VAL mode, the code generated for this clause might be

```
1   CLA    a
2   NEQ    JCC L1
3   CLA    b
4   NEQ    JCC L3
5   LD     c
6   JCC    L4
7 L3:    LD     d
8 L4:    JCC    L2
9 L1:    LD     e
10 L2:   ...
```

The problem that arises is apparent at line 6 above. At line 6 there is an unconditional jump to L4, while at L4 there is an unconditional jump to L2. This is somewhat inefficient and very costly in the presence of loops. To prevent this, the code generator checks each labelled instruction to see if it is a jump. If it is, all jumps to that label are patched to refer to the destination of the labelled jump instruction. Thus the jump at line 6 would be changed to jump to L2.

6.3 LABEL VARIABLES AND INTERSEGMENT JUMPS

Jumps out of segments present a more difficult problem at run-time. As an example, consider the routine below.
(STATIC v:=VEC (5) #create a vector of 5 elements#

\[ t:=\text{PROC}(m,n): \]

\[ (\text{IF } n \text{ NE } 0 \text{ THEN} \]
\[ v[n]:=11; \ t(m,n-1) \]
\[ \text{ELSE GOTO } v[m] \FI; \]
\[ 11: \text{print } (n) \]
\[ ); \ t(3,5) \]

). Notice that at each recursion level of routine \( t \), the label \( 11 \) is assigned to an element of the vector \( v \). One would expect the elements of the vector \( v \) to be different, each reflecting the number \( n \) corresponding to the level of recursion. A little study should convince one that the proper result of the call \( t(3,5) \) is the output of the numbers 3, 4, and 5.

In order for this to work properly, a label variable must not only reflect the location of the label it represents, but also the proper environment at the time the label was evaluated in VAL mode. Consider first of all the jump destination. Since segments are required to be identified logically, the label variable should contain the logical name of the segment. In addition, it must contain information necessary for finding the proper instruction in the segment. To do this, the entry table in the segment is used. Whenever a label is evaluated in VAL mode, an entry is created for it in the entry table. A label variable contains the index into this table which
identifies the control word describing the labelled instruction.

To restore the environment, the label variable must contain information about the status of the two dynamically varying regions in the NEW program: the stack and the local vector region. To do this, a copy of the stack pointer and the index of X13 in the local vector region is saved. Obviously, the information required in a label variable is much too large to fit in one word. Since it is required that all values be single word items, a label variable is represented by an address word. The address word describes a two word vector containing the necessary information. This vector is created in the local vector region at run-time. To execute a jump using a label variable, that variable must be interpreted. This is done by the GOTOSOLVER routine which restores the proper environment, finds the segment, and executes the jump.
CHAPTER VII

THE TRANS RUTINE

TRANS is the main driver for the code generator. Once GETWGT has been used to assign weights to the tree nodes, TRANS is called to generate the object code. As was suggested by the Sethi and Ullman algorithm, TRANS is a recursive procedure. When generating the code for a particular node, it will call itself to generate the code for the operands of that node. As was seen before, the weights of the nodes determine the order in which operands are evaluated in the case of arithmetic expressions. This is also true for the more complicated data accession nodes: subscription and field selection. For control nodes, however, the order is fixed by the semantics, so the best we can hope for is that control variables be in registers for faster access.

The input to TRANS is an operand, which may be a tree pointer or a terminal. Terminal operands are used to represent variables and constant denotations: literals, numeric constants, strings, and character constants. Literals are addressed immediately, using the \texttt{±N} field, while other constants are accessed through the data vector. When a constant is encountered, its value is built in the data vector, and since this region is constant, a check is made for duplicate entries in order to keep this region as compact as possible. All terminal operands are directly addressable. To obtain the address, a routine called \texttt{GENL} is used. If the address is not immediately available, as
in relative addressing to the data vector, an invalid Y-field value is used to distinguish the instruction. When the segment is complete, these instructions are patched. Using the information from GENY and the mode of elaboration, TRANS can generate the instructions required to load the value to the proper register, or in the case of GOTO mode, to generate a jump.

The CODE routine is used to actually load instructions into the object buffer. In addition to the instruction fields, the operand is passed so that symbolic names may be printed. The principal activity in CODE is to produce a pseudo-assembly language listing of the object code. In addition, a small amount of local optimization is done. For example, stacking operations and loads are consolidated into Save and Fetch instructions. Store option bits are set when applicable and the check for cascaded jumps is made.

When a tree pointer is passed to TRANS, the tree node is considered and the STYPE and FTYPE fields used to locate the relevant generator. Appendix C contains a listing of the various tree nodes and a short description of how they are handled by TRANS. In many cases, all nodes with a particular STYPE may be handled by the same routine, for example, the set of commutative operators.

In generating code for some constructs, it is necessary, for purposes of efficiency, to have certain information about its operands. Several routines are provided for
this purpose. They are INREGISTER, ISADDRESSABLE, and
EVALCONST. INREGISTER takes an operand as its parameter
and determines if it resides in a register; if so, the
register number is returned. ISADDRESSABLE checks to see
if an operand is addressable. Some fairly complicated
expressions may be calculated by the addressing rule, thus
making this routine useful. For example, A+3 and A[3] are
both immediately addressable if A is in a register. In
the first case, A must be an integer, while in the second
case, it must be an address. EVALCONST tests its operand
to see if it is a constant; if it is, the expression is
evaluated at compile-time and the value stored in a common
location.

In order to get the value of an operand into a register,
one merely calls TRAns using that operand as a parameter.
The elaboration mode and the target register must be set
properly. In some instances, however, it is not required
that the actual value be loaded, but only that the expres¬
sion be reduced until it is addressable. In this case,
the routine MAKEADDRESSABLE is used. This routine per¬
forms just enough computation to make the operand address¬
able and then returns the addressing information.

In determining how to handle a particular construct
the routines mentioned above, along with the weight informa¬
tion are used in making decisions about the order in which
operands are calculated. If temporary results are needed,
an attempt is made to place them in registers. If no
registers are available, the stack is used for this purpose. One trick that is often useful in VAL mode is to use the target register, which is always free, as a temporary register.

The details of the various generators are not given here, since the organization of the code generator is more important. However, one should realize that enough information has been made available to permit making good decisions in generating the code. This information is not particularly useful if one is merely concerned with generating workable object code, however, the goal here is to obtain reasonable efficiency as cheaply as possible.
SUMMARY AND CONCLUSION

In this thesis, I have presented the organizational structure of a code generator for the R2 using a particular source language. Many of the techniques used here are particularly well-suited to the R2 and could be used as a guideline in writing a code generator for other source languages. One possible area for future research might be a universal code generator which could be used for any language on the R2.

There are, however, significant areas of code generation which have not been touched on in this thesis. These are in the area of global optimization. All optimization done by the current code generator is of a local nature. It still remains to be seen how efficiently one can write system programs in NEW. Whenever a high-level language is substituted for assembly language, the costs, in terms of efficiency, should be weighed against the convenience of being able to program at a much less detailed level. A major goal in the design of NEW was to provide a powerful language facility with constructs closely related to particular R2 features so as to insure efficient object code for each construct. In this regard, I believe the language is successful. However, a large burden still remains with the programmer, who must insure that his programs are efficient at the algorithmic level, in order to obtain efficiency at the object level. Just how much
more optimization should be added to the compiler is another area for future research.

In the course of this research, several deficiencies have been discovered, both in the NEW language and in the R2. I feel that the recognition of these problems is perhaps the greatest benefit derived from this research. These problems are summarized below.

1) Register Declarations -- Although the current declarations only provide for two classes of storage, there is actually another very important class: register storage. Register declarations should be added to the NEW language to indicate those variables which should be in registers when used as operands. Since good register allocation schemes are practically non-existent, this would give the programmer the ability to notify the compiler that certain variables are 'important' and should be kept in registers. In addition, the ambiguities which arise in the addressing rule concerning the use of chained address words could be avoided. Finally, this would permit a more formal description of the semantics of NEW.

2) Types -- NEW is currently a typeless language. While this is often a convenience, it inhibits the optimization process and does not allow any validity checking at compile-time. Unfortunately,
the lack of any typing in NEW leaves the programmer responsible for providing the proper number of register parameters in a call. This particular problem alone could cause some serious debugging problems.

3) Boolean values in the R2 -- There is no means provided in the R2 to set the boolean condition code when a value is loaded to a register. It is suggested that an instruction similar to Clear and Add be added to perform this function. In addition, it is very difficult to obtain a boolean value based on the condition code setting. One proposed solution would be an additional instruction, similar to Jump on Condition Code, which instead of jumping would cause a 1 or 0 to be stored to the Y-operand. This one instruction would replace the three instructions currently used by the code generator for this purpose.

4) The MOD option -- The MOD option, which appears in all arithmetic and boolean instructions, is virtually useless. A better use for this bit might be to combine it with the current store option bit to produce a four variant field. This field would provide for normal operation, store option, a test option, and a spare. This would eliminate the need for the current Test instruction. In addition, the five boolean instructions could be replaced by three other
instructions: Boolean, Boolean and Store, and Boolean and Test. The X-field of these instructions could be used to choose among the sixteen boolean functions.

5) GET and PUT instructions -- The interpretation of the GET and PUT instructions is awkward to use since the accumulator is used to hold the value loaded or stored. It is suggested that two new instructions, similar to GET and PUT, be added to the machine. The interpretation would be as follows: the address word being indexed would reside in the accumulator, while the X-field would select the source or destination register.

6) Parameter passing -- There seems to be no satisfactory scheme for passing parameters to subroutines in the R2. The NEW compiler does this by stacking the parameters and copying the stack pointer into X14. This copy is then used as the parameter pointer. This has the disadvantage that the stack cannot be segmented. Several methods have been proposed to alleviate the situation, but no good solution is available as yet.
APPENDIX A

A COLLECTED SYNTAX FOR NEW

0) MODE base: MODE identifier; MODE denotation; MODE skip.

1) VARIABLE identifier: tag.

2) NONREF denotation: boolean; number; string; label denotation.

3) boolean: true symbol; false symbol.

4) number: integer; real; pattern; ascii.

5) integer: digit sequence.

6) real: integer, fraction, exponent option; integer, exponent.

7) fraction: point symbol, integer.

8) exponent: at symbol, plusminus option, integer.

9) plusminus: plus symbol; minus symbol.

10) pattern: RADIX bit string.

11) RADIX bit string: RADIX radix, quote symbol, RADIX token sequence, quote symbol.

12) two radix: two symbol.

13) four radix: four symbol.

14) eight radix: eight symbol.

15) sixteen radix: one symbol, six symbol.

16) two token: zero symbol; one symbol.

17) four token: two token; two symbol; three symbol.

18) eight token: four token; four symbol; five symbol; six symbol; seven symbol.
19) sixteen token: eight token; eight symbol; nine symbol; upper a symbol; upper b symbol; upper c symbol; upper d symbol; upper e symbol; upper f symbol.
20) ascii: single quote symbol, character token sequence option, single quote symbol.
21) string: quote symbol, character token sequence option, quote symbol.
22) label denotation: tag.
23) MODE skip: skip symbol.
24) MODE primary: MODE base; MODE subscription; MODE call; MODE generator; MODE package.
25) VARIABLE subscription: value primary, slicer.
26) slicer: sub symbol, index list, bus symbol.
27) index: value unit; initial zero symbol, offset option.
28) offset: plus symbol, value unit.
29) VARIABLE generator: vec symbol, bounds part; locvec symbol, bounds part.
30) bounds part: open symbol, bound list, close symbol, bounds part.
31) bound: value unit; value unit, colon symbol, value unit.
32) MODE secondary: MODE primary; MODE partial word designator.
33) VARIABLE partial word designator: VARIABLE secondary, point symbol, sub symbol, value unit, colon symbol, value unit, bus symbol.
34) constant partial word designator: constant secondary, point symbol, sub symbol, constant unit, colon symbol, constant unit, sub symbol.

35) MODE tertiary: MODE secondary; MODE ADIC formula.

36) MODE PRIORITY formula: LMODE PRIORITY operand, LMODE and RMODE in MODE PRIORITY operator, RMODE PRIORITY plus one operand.

37) MODE PRIORITY operand: MODE PRIORITY formula; MODE PRIORITY plus one operand.

38) MODE PRIORITY TEN plus one operand: MODE monadic operand.

39) MODE monadic operand: MODE monadic formula; MODE secondary.

40) MODE monadic formula: RMODE in MODE monadic operator, RMODE monadic operand.

41) MODE quaternary: MODE tertiary; MODE assignation.

42) VARIABLE assignation: destination, becomes symbol, source.

43) destination: reference tertiary.

44) source: value unit.

45) MODE unit: MODE quaternary; MODE routine; MODE repetition; MODE jump.

46) VALPAR routine: proc symbol, formals option, colon symbol, value unit.

47) formals: open symbol, register params option, normal part option, close symbol.

48) register params: tag list.
49) normal part: go on symbol, param vec option, normal params; go on symbol, param vec.

50) param vec: tag, colon symbol.

50a) normal params: tag list.

51) VARIABLE repetition: index part, before part, controlled part, after part.

52) index part: for indexer; forall indexer.

53) for indexer: counter option, start option, step option, stop option.

54) counter: for symbol, tag.

55) start: from symbol, value unit.

56) step: by symbol, value unit.

57) stop: to symbol, value unit.

58) forall indexer: forall symbol, tag list, in symbol, value unit list.

59) before part: test option.

60) after part: test option.

61) test: while symbol, value unit; until symbol, value unit.

62) controlled part: do symbol, value unit.

63) VARIABLE jump: goto symbol, value unit.

64) declaration: local declaration; static declaration; external declaration; define declaration.

65) local declaration: local symbol, new variable list.

66) static declaration: static symbol, new variable list.

67) new variable: tag, initialization option.

68) initialization: becomes symbol, value unit.
69) external declaration: external symbol, new external list.
70) new external: external name option, tag, initialization option.
71) external name: string, colon symbol.
72) define declaration: define symbol, new macro list.
73) new macro: tag, macro parameters option, equal symbol, macro text.
74) macro parameters: open symbol, tag list, close symbol.
75) MODE package: MODE closed clause; MODE conditional clause; MODE case clause; MODE collateral clause.
76) MODE closed clause: elem symbol option, open symbol, MODE serial clause, close symbol.
77) VARIABLE serial clause: declaration prelude sequence option, chain of VARIABLE clause trains.
78) declaration prelude: declaration, go on symbol; value unit, go on symbol, declarition prelude.
79) chain of VARIABLE clause trains: VARIABLE clause train; VARIABLE clause train, completer, chain of VARIABLE clause trains.
80) VARIABLE clause train: VARIABLE statement; value statement, go on symbol, VARIABLE clause train.
81) VARIABLE statement: label sequence option, VARIABLE unit.
82) label: tag, color symbol.
83) completer: completion symbol, label.
84) VARIABLE conditional clause: open symbol, value serial clause, then else symbol, VARIABLE serial clause, VARIABLE else part option, close symbol.
85) VARIABLE else part: then else symbol, VARIABLE serial clause.
86) constant conditional clause: open symbol, constant unit, then else symbol, constant unit, then else symbol, constant unit, close symbol.
87) VARIABLE case clause: open symbol, value serial clause, then else symbol, VARIABLE unit list, VARIABLE out part option, close symbol.
88) VARIABLE out part: then else symbol, VARIABLE unit.
89) constant case clause: open symbol, constant unit, then else symbol, constant unit list, then else symbol, constant unit, close symbol.
90) VARIABLE collateral clause: open symbol, VARIABLE unit list proper, close symbol.
91) VARIABLE call: value primary, open symbol, mark option, register actuals option, normal actuals option, close symbol.
92) mark: constant unit, colon symbol.
93) register actuals: value unit list.
94) normal actuals: go on symbol, parameter unit list.
METAPRODUCTION RULES

a) ADIC: PRIORITY; monadic.
b) EIGHT: SEVEN plus one.
c) EMPTY:.
d) FIVE: FOUR plus one.
e) FOUR: THREE plus one.
f) LMODE: MODE.
g) MODE: VARIABLE; constant.
h) NINE: EIGHT plus one.
i) NONREF: parameter; value; constant.
j) NOTION list: NOTION; NOTION, comma symbol, NOTION list.
k) NOTION list proper: NOTION, comma symbol, NOTION list.
l) NOTION option: NOTION; EMPTY.
m) NOTION sequence: NOTION; NOTION, NOTION, NOTION sequence.
n) NUMBER: one; NUMBER plus one.
o) PRIORITY: priority NUMBER.
p) RADIX: two; four; eight; sixteen.
q) RMODE: MODE.
r) SEVEN: SIX plus one.
s) SIX: FIVE plus one.
t) TEN: NINE plus one.
u) THREE: TWO plus one.
v) TWO: one plus one.
w) VALPAR: value; parameter.
x) VARIABLE: reference; parameter; value.
APPENDIX B

THE R2 INSTRUCTION SET

FORMAT

R2 instructions are thirty bits in length and are stored two to a word. The form of the instruction is

\[ A \times/V \quad \text{OP} \quad Y \pm N \]

The instruction fields have the following meaning:

- **A** The a-bit. A one bit field used by the addressing rule to modify the interpretation of the Y field.
- **X/V** The X or Variant field. This four bit field is used either to specify one of sixteen general-purpose registers, or to choose an instruction variant.
- **OP** The OP code. A six bit field used to specify the function of the instruction.
- **Y** The Y field. A four bit field which specifies either a general-purpose register, or an addressing variant.
- **±N** The ±N field. This fourteen bit field provides for immediate addressing of small integers, absolute core addressing, and an offset for indexing.

THE ADDRESSING RULE

Normally an instruction in the R2 obtains one of its
operands from a register, while the second operand is obtained via the addressing rule. The addressing rule uses the A, Y, and $\pm N$ fields to determine the operand in the following manner:

**A=0** The Y field is used to select a register. The $\pm N$ field is considered to be an integer in one-complement. If the $\pm N$ field is all ones (-0), then the operand is the contents of the register selected by the Y field. Otherwise, if the Y register contains an integer, then the $\pm N$ field is added to that integer and the result is the required operand. Finally, if the Y register is an address word, then the $\pm N$ field is taken to be an index and an indexed fetch is initiated. The result of the fetch is the required operand.

**A=1** In this case, the Y field is used to specify an addressing variant as follows:

**Y=0** The $\pm N$ field is taken as the operand. This is immediate addressing.

**Y=1** The $\pm N$ field is taken to be an unsigned fourteen bit integer address. This address is used to find an absolute core location, the contents of which is the operand.

**Y=2** The $\pm N$ field is added to the CN (control number). If the instruction is a jump, then the result represents the address of the jump destination. Otherwise, the
high-order 20 bits of the result are used as a memory address. The contents of this address is the operand.

**Y=4** The instruction must be a jump. The $+N$ field is used to select a word as in the $Y=2,3$ case above. This word must be an address word describing a vector of control words. The accumulator, $U$, is used as an index into this vector; the resulting control word locates the destination of the jump. This option is used to implement the CASE expression in NEW.

**CONVENTIONS**

In the following, $B$ will be understood to be the $Y$ or second operand fetched by the address rule. The accumulator will be denoted by $U$, and its extension by $\bar{R}$.

**INDEXING**

Address words in the R2 are used to describe one-dimensional vectors of contiguous core locations. An address word has three fields used by the indexing process: LOC, IO, and LENGTH. The LOC field holds the address of the first physical word of the vector. IO is the initial index, and LENGTH the length of the vector. When an address word is indexed, the address of the required word is given by the formula
LOC\[B-I0,\]

where B is the index. The hardware insures that

0 ≤ B-I0 ≤ LENGTH

If this condition is not met, an error trap occurs.

An address word may be absolute or relative. The LOC field of an absolute address word contains the actual address of the vector in memory. In a relative address word, however, the LOC field is considered to be a signed offset from the core address of the address word itself. When a relative address word is brought to a register, it is de-relativized, i.e. the LOC field is made absolute. Similarly, when the address word is stored, it is relativized and the LOC field is made relative again.

STACKING

The register, X0, is used by the R2 to point to a stack in memory. The stack is a contiguous vector in memory which is bounded at either end by partition words. These partition words are used to prevent the stack from growing indefinitely. X0 can be considered to be an address word with initial index 0. When indexing through X0, however, index 1 yields the 'top' of the stack. Index 0 is used to cause a new element to be 'pushed' onto the stack or 'popped' from the stack. The stack is also used to store subroutine return links when a Jump and Set Mark instruction is executed.
OPERATIONS ON NUMERIC WORDS

The instructions in this class provide for arithmetic and boolean operations. The first operand is always assumed to be in U. The X field is used as an instruction variant. In all instructions of this class except NOCONV, USRGB, and TEST, the high-order two bits of the X field are used for the STORE and MOD option, respectively. The STORE option bit, if set, causes the result of the operation (which is in U), to be store back to the source of the second operand. This is useful in implementing the operators in NEW which combine operations with assignations. The MOD option bit, if set, causes the Y address word to be MOD'ed by 1 upon completion of the operation. It is of dubious utility. The low-order two bits are interpreted differently by the various instructions in this class.

ARITHMETIC OPERATIONS

There are four instructions in this group. They are listed by operation and the variants performed by the third and fourth bits in the X field.

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>X₃ X₄</th>
<th>DESCRIPTION OF RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0 0</td>
<td>U+B</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>U-B SUB</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>-(U+B)</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>B-U BUS</td>
</tr>
<tr>
<td>MPY</td>
<td>0 0</td>
<td>U*B</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>B*U</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>-U*B</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>-B*U</td>
</tr>
<tr>
<td>INSTRUCTION</td>
<td>X₃ X₄</td>
<td>DESCRIPTION OF RESULT</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>DIV</td>
<td>0 0</td>
<td>U/B</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>B/U VID</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>-U/B -DIV</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>-B/U -VID</td>
</tr>
<tr>
<td>REM</td>
<td>0 0</td>
<td>Rem (U/B)</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>Rem (B/U) MER</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>-Rem (U/B) -REM</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>-Rem (U/B) -MER</td>
</tr>
</tbody>
</table>

**BOOLEAN OPERATIONS**

This class includes five full-word bitwise boolean operations and the Extract/Mask Compare instruction. X₃ and X₄ are interpreted as a number which selects four variants. In the case of the first five instructions, the variants are as follows:

- **0 0**: The normal function is performed.
- **0 1**: The result is complemented.
- **1 0**: The operation is performed, but U is left unchanged. The result of the operation is reflected in the CC.
- **1 1**: Only used in the AND and OR instructions.

The five boolean instructions and their variants are as follows:

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>X₃ X₄</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0 0</td>
<td>Logical product</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>NAND</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>U is complemented</td>
</tr>
</tbody>
</table>
The Extract/Mask Compare instruction uses R as a mask to allow extracting bit fields from B and comparing bits in U and B. If $x_3$ is 0 then the Extract operation is selected, otherwise the Mask Compare operation is selected. $x_4$ determines whether R or its complement will be used as the mask. The options are:

<table>
<thead>
<tr>
<th>$x_3$</th>
<th>$x_4$</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Replace bit i of U with bit i of B if bit i of R is 1.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Replace bit i of U with bit i of B if bit i of R is 0.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Bit i of U is set to 1 if the i-th bits of U and B are different and bit i of R is 0.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Bit i of U is set to 1 if the i-th bits of U and B are different and bit i of R is 1.</td>
</tr>
</tbody>
</table>

### Instruction Table

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>Logical sum</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>NOR</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>B complement is loaded to U.</td>
</tr>
<tr>
<td>SYD</td>
<td>0</td>
<td>0</td>
<td>Symmetric difference</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Equivalence</td>
</tr>
<tr>
<td>IMP</td>
<td>0</td>
<td>0</td>
<td>Implication</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>U AND NOT B</td>
</tr>
<tr>
<td>RIMP</td>
<td>0</td>
<td>0</td>
<td>NOT B OR U</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>B AND NOT U</td>
</tr>
</tbody>
</table>
NUMBER CONVERSION (MCONV)

This instruction provides for converting from real to integer and from integer to real. If X₄ is 1 the STORE option is applied. If X₃ is 0 then B is converted to integer and brought to U, otherwise, B is converted to real and brought to U.

USRGB

The mnemonic for this instruction stands for U,S,R Grab Bag. The instruction provides for a number of useful operations. Actually the S is an anomaly, since there is no S register. The S register held the second operand in the R1. The instruction uses the X field to determine the function performed. These are as follows:

<table>
<thead>
<tr>
<th>X field</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>ZSTO</td>
</tr>
<tr>
<td>0001</td>
<td>ISTO</td>
</tr>
<tr>
<td>0010</td>
<td>-ZSTO</td>
</tr>
<tr>
<td>0011</td>
<td>-ISTO</td>
</tr>
<tr>
<td>0100</td>
<td>STR</td>
</tr>
<tr>
<td>0101</td>
<td>LDR</td>
</tr>
<tr>
<td>0110</td>
<td>SUR</td>
</tr>
<tr>
<td>0111</td>
<td>XUR</td>
</tr>
<tr>
<td>1000</td>
<td>CLA</td>
</tr>
<tr>
<td>1001</td>
<td>NEG</td>
</tr>
<tr>
<td>1010</td>
<td>ABS</td>
</tr>
<tr>
<td>1011</td>
<td>-ABS</td>
</tr>
</tbody>
</table>

- ZSTO: Store a zero to Y
- ISTO: Store a one to Y
- -ZSTO: Store a -0 to Y
- -ISTO: Store a -1 to Y
- STR: Store R to Y
- LDR: Load R from B
- SUR: Shuffle U and R. U goes to R and B goes to U.
- XUR: Exchange U and R
- CLA: Clear and Add
- NEG: -B goes to U
- ABS: ABS B goes to U
- -ABS: -ABS B goes to U
<table>
<thead>
<tr>
<th>X field</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>CLA (S)</td>
</tr>
<tr>
<td>1101</td>
<td>NEG (S)</td>
</tr>
<tr>
<td>1110</td>
<td>ABS (S) STORE</td>
</tr>
<tr>
<td>1111</td>
<td>NBS (S)</td>
</tr>
</tbody>
</table>

TEST

The test instruction is used to set the CC for various conditions. The X field specifies an operation to be performed. The result of that operation is reflected in the CC, but U is left unchanged.

<table>
<thead>
<tr>
<th>X field</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>U+B</td>
</tr>
<tr>
<td>0001</td>
<td>U+ABS B</td>
</tr>
<tr>
<td>0010</td>
<td>U+B</td>
</tr>
<tr>
<td>0011</td>
<td>U+ABS B</td>
</tr>
<tr>
<td>0100</td>
<td>U-B</td>
</tr>
<tr>
<td>0101</td>
<td>U-ABS B</td>
</tr>
<tr>
<td>0110</td>
<td>ABS U-B</td>
</tr>
<tr>
<td>0111</td>
<td>ABS U-ABS B</td>
</tr>
<tr>
<td>1000</td>
<td>U*ABS B</td>
</tr>
<tr>
<td>1001</td>
<td>Extract</td>
</tr>
<tr>
<td>1010</td>
<td>Extract on NOT B</td>
</tr>
<tr>
<td>1011</td>
<td>Extract on NOT R</td>
</tr>
<tr>
<td>1100</td>
<td>U/B</td>
</tr>
<tr>
<td>1101</td>
<td>Mask Compare</td>
</tr>
<tr>
<td>1110</td>
<td>Rem (U/B)</td>
</tr>
<tr>
<td>1111</td>
<td>Mask Compare on NOT R</td>
</tr>
</tbody>
</table>
A NOTE ON CODE GENERATION

As one can see, the instructions for numeric operations are very rich in possibilities for local optimization. The STORE option provides an easy implementation of the operator-and-assignation operators in NEW. The monadic operators ABS, NOT, and - are almost always absorbed into other instructions. Since the TEST instruction is much more powerful than the more common Compare instruction found in most machines, one can test conditions such as \( a + b \ LE 0 \) with fewer instructions than might normally be required. Of course, in performing these operations, one must pay the price of a slower compiler, since the various options must be constantly checked for.

SHifting

There are two instructions in the R2 which provide for shifting: SHA and SHB. The second operand determines the number of places to shift. If this is negative, then the direction of the shift is reversed. In any case, the shift length is the absolute value of B modulo 128.

SHA -- U and R are both shifted. The X field controls the shift as follows:

\[
\begin{align*}
X_1 & \quad 0 \quad \text{Shift U Right} \\
& \quad 1 \quad \text{Shift U Left} \\
X_2 & \quad 0 \quad \text{Zero fill vacated bits in U} \\
& \quad 1 \quad \text{Spill from R fills vacated bits} \\
X_3 & \quad 0 \quad \text{Shift R Right} \\
& \quad 1 \quad \text{Shift R Left}
\end{align*}
\]
X₄ 0  Zero fill vacated bits in R
     1  Spill from U fills vacated bits

SHB -- The X field of the instruction determines the operation as follows:

<table>
<thead>
<tr>
<th>X (octal)</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>U logical left</td>
</tr>
<tr>
<td>1</td>
<td>U logical right</td>
</tr>
<tr>
<td>2</td>
<td>R logical left</td>
</tr>
<tr>
<td>3</td>
<td>R logical right</td>
</tr>
<tr>
<td>4</td>
<td>U, R arithmetic left</td>
</tr>
<tr>
<td>5</td>
<td>U, R arithmetic right</td>
</tr>
<tr>
<td>6</td>
<td>--NOT USED--</td>
</tr>
<tr>
<td>7</td>
<td>--NOT USED--</td>
</tr>
<tr>
<td>10</td>
<td>Bit count</td>
</tr>
<tr>
<td>11</td>
<td>Bit Count Accumulate</td>
</tr>
<tr>
<td>12</td>
<td>Zero Count</td>
</tr>
<tr>
<td>13</td>
<td>Zero Count Accumulate</td>
</tr>
<tr>
<td>14</td>
<td>Most significant bit</td>
</tr>
<tr>
<td>15</td>
<td>Least significant bit</td>
</tr>
<tr>
<td>16</td>
<td>Most significant 0</td>
</tr>
<tr>
<td>17</td>
<td>Least significant 0</td>
</tr>
</tbody>
</table>

DATA MANIPULATION

The data manipulation instructions provide for moving data to and from registers and for extracting and changing the various fields in the R2 words.
LD Load the X register with the Y operand.
STO Store the X register to the Y operand.
CBL Chain Break Load. This is the same as load, but no chaining is allowed.
CBS Chain Break Store. This is the same as store, but no chaining is allowed.
LIZ Load Initial Zero. This is the same as load, except the initial index, 10, of the Y field is assumed to be zero.
SIZ Store Initial Zero. Similar to LIZ.
SVF Save and Fetch. This is similar to load, but the X register is pushed onto the stack before the load takes place, thus saving the old contents.
XCH Exchange. The X register and the second operand are exchanged.

FIELD MANIPULATION

There are five instructions used for obtaining and changing the values of the various fields in R2 words.

XFA, XFB The extract field instructions. Since there are a possible 32 fields, two instructions are required. XFA is used to select the first 16, while XFB selects the final 16. The selection is done by the X field of the instruction and the field extracted is brought to U. If the field is a signed
field, the sign is propagated. Otherwise
U is zero filled to the left.

RFA, RFB The replace field instructions. These
instructions are similar to XFA and XFB.
The field in the second operand is replaced
by the proper number of low-order bits from U.

TAG The TAG instruction is a special immediate
form of Replace Field. The X field and
the a-bit are concatenated to allow selec-
tion of all 32 fields. The Y field selects
a register whose field is to be changed.
The $+$N field contains the replacement data.

The fields which may be selected by the above instruc-
tions are the following:

<table>
<thead>
<tr>
<th>Field Number</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Left-hand instruction</td>
</tr>
<tr>
<td>1</td>
<td>Right-hand instruction</td>
</tr>
<tr>
<td>2</td>
<td>X field</td>
</tr>
<tr>
<td>3</td>
<td>OP field</td>
</tr>
<tr>
<td>4</td>
<td>Y field</td>
</tr>
<tr>
<td>5</td>
<td>$+$N field</td>
</tr>
<tr>
<td>6</td>
<td>Write Lockout Bit</td>
</tr>
<tr>
<td>7</td>
<td>Software Tags</td>
</tr>
<tr>
<td>8</td>
<td>Hardware Tags</td>
</tr>
<tr>
<td>9</td>
<td>Restricted Access Bit</td>
</tr>
<tr>
<td>10</td>
<td>Indirect Tags</td>
</tr>
<tr>
<td>11</td>
<td>Presence Bit</td>
</tr>
<tr>
<td>Field Number</td>
<td>Field Name</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>12</td>
<td>Length field of address word</td>
</tr>
<tr>
<td>13</td>
<td>Initial Index</td>
</tr>
<tr>
<td>14</td>
<td>Location field of address word</td>
</tr>
<tr>
<td>15</td>
<td>Location field of control word</td>
</tr>
<tr>
<td>16</td>
<td>Condition Code</td>
</tr>
<tr>
<td>17</td>
<td>Software Tag of Instruction</td>
</tr>
<tr>
<td>18</td>
<td>A-bit of Instruction</td>
</tr>
<tr>
<td>19</td>
<td>Sign bit of mantissa</td>
</tr>
<tr>
<td>20</td>
<td>Exponent</td>
</tr>
<tr>
<td>21</td>
<td>High-order five bits of initial index</td>
</tr>
<tr>
<td>22</td>
<td>Second 5 bits of Initial index</td>
</tr>
<tr>
<td>23</td>
<td>Mode field of control word</td>
</tr>
<tr>
<td>31</td>
<td>Parity bit</td>
</tr>
</tbody>
</table>

**BLOCK MANIPULATION**

There are two instructions which are useful in moving blocks of storage from one memory location to another. The \( X \) field must be an address word which is MODed by 1 after the instruction is complete. This is used to describe the destination. The addressing rule is used to fetch the data which is transferred through the \( X \) address word. Two instructions are provided which handle address words in the block differently.

**MOV** If an address word is fetched by the \( Y \) operand it is possibly de-relativized and then relativized during the transfer. The assumption is
that the relative address word involved points to a location outside the block being moved. If it does not, then the MOV instruction will leave it pointing to the old block.

**BLT** The Block Transfer instruction is similar to the Move instruction, except relative address words are not relativized or de-relativized. The assumption is that any relative address words describe words in the block being transferred. If not, the address word will be wrong after the transfer is complete.

Care must be taken in the use of these instructions to insure that the proper one is used.

**BRANCHING INSTRUCTIONS**

All jumps take the Y address as their destination. The X field either specifies a register, a mark field, or a condition. When the X field specifies a condition, as in the JPT and JCC instructions, the Bits of the X field from left to right denote the values 0-3. The test is made against the field being tested to see if it has the value represented by a bit in the X field. Thus, any or all of the values may be tested simultaneously.

**JSM** The X field specifies a mark field. A control word is constructed which describes the machine state and contains the mark specified. This word is stacked and the stack length set to zero.
JSL  Jump and Set Link. This is similar to Jump and Set Mark, except the X field chooses a register to store the return control word in. The stack is unaffected.

JCC  Jump on Condition Code. The X field specifies a test on the Condition Code register. If any of the values match, the jump is taken.

JPT  Jump on Programmer Tags. This is similar to JCC but the software tag register is used rather than CC.

JL   Jump on Last. The X-register is tested to see if it is an address word with length field equal to zero. If it is, the jump is taken. Otherwise, the X address word is MODed by 1.

JnL  Jump is not Last. This is the same as JL, except the jump is taken if the length field is not equal to zero. The X address word is MODed as above.

JLT  Jump Less Than. The X register must contain an integer. If that integer is less than zero, then the jump is taken, otherwise it is decremented and the next instruction taken.

JGE  Jump Greater or Equal. This is similar to JLT, but the jump is taken if the X register is greater or equal to zero.
RET Return to Mark. This is a special jump used to return after a JSM has been executed. The stack is searched for a return control word with a mark matching the Y field or related to it as specified by the X field. If the X field is zero, the search is for a mark less than the Y field. If the X field is 1, the search is for a mark greater than the Y field.

ADDRESS MANIPULATION

These instructions are used to manipulate address words in a control fashion. Included in this set are some special data manipulation instructions.

DOT The second operand is used as an index to the X register and the result is brought to the X register.

GET This is similar to DOT, except the result is brought to U.

PUT This is similar to GET, but U is stored to the resulting address.

MOD The second operand must yield an integer which is simultaneously added to the X register LOC field and subtracted from the LENGTH field. If the resulting LENGTH is negative, an error trap occurs.

LIM This is similar to MOD, but the integer simply replaces the LENGTH field if it is currently less than or equal to the previous LENGTH field. LIM shortens an array from the high-index end, while MOD shortens it from the low-
MEM Membership test. The X address word is tested to see if it describes a member of the vector described by the address word fetched by the addressing rule. If it does, the index of X in Y replaces U. The CC is set depending on whether the resulting index is the initial index or whether the MEM test is true or false.

THE CONTROL INSTRUCTION (CTR)

This instruction is used to interrogate various registers in the machine and to set them by the program. The X field is used for a variant.

<table>
<thead>
<tr>
<th>X (octal)</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set the CC from U</td>
</tr>
<tr>
<td>1</td>
<td>Set the MODE from U</td>
</tr>
<tr>
<td>2</td>
<td>Set the Software Tags from U</td>
</tr>
<tr>
<td>3</td>
<td>Set bits in MODE corresponding to bits in U to 1.</td>
</tr>
<tr>
<td>4</td>
<td>Clear bits in MODE corresponding to bits in U.</td>
</tr>
<tr>
<td>5</td>
<td>Load U from the CC</td>
</tr>
<tr>
<td>6</td>
<td>Load U from the MODE</td>
</tr>
<tr>
<td>7</td>
<td>Load U from the software tag register.</td>
</tr>
</tbody>
</table>
APPENDIX C

TREE NODES IN NEW

The syntax tree for NEW controls the action performed by the code generator. The following is a description of the various control nodes and a short description of the use of the operands.

BECOMES Assignations. The first operand is the left-hand side, while the second operand is the right-hand side.

DO Repetitions. There are four operands:
1) A pointer to the indexer, if any. The indexer must be a FOR or a FORALL node.
2) The before test option.
3) The controlled unit.
4) The after test option.

IF Conditionals. There are three operands:
1) The choice clause.
2) The true arm.
3) The false arm, if any.

CASE Cases. There are three operands:
1) Index unit.
2) List of cases. This must be a pointer to a COMMA node.
3) Default or OUT condition.
CALL
Routine Call. Four operands:
1) The mark. This must be a constant.
2) The routine.
3) Normal parameters.
4) Register parameters.

LABDEF
Label definition. The label number is placed using COMPLAB. The first operand is this number. The second operand points to the labelled unit.

SERCL
Serial Clause. There are three operands:
1) A pointer to the symbol table information for declarations in this serial clause.
2) Execution units. This must describe a unit or a clause train.
3) The number of declarations described by 1.

CLTRAIN
Clause train. There are three operands:
1) Chain of units separated by GOON nodes.
2) The value producing unit.
3) A pointer to the next clause train, if any.

If operand 3 describes a clause train, then a completer label is generated and jumped to after the completion of code for the various units. VOID mode is used for the units described by
the first operand.

**GO ON** Go on symbol. This node is used to chain together the units in a clause train. The first operand is a unit, while the second operand describes the final unit or another GO ON node.

**COLLAT** Collateral clause. This node describes a list of units which will be evaluated and placed in a vector created in the LOCVEC region. Each unit is elaborated in the same mode.

**ELEM** Elementary node. This node marks a serial clause in which no rearrangement of elements is allowed. In particular formulas may not be reordered by the Sethi and Ullman algorithm.

**GOTO** Goto or Jump. The operand is elaborated in GOTO mode resulting in a jump to the given label.

**RETL** Return to mark less than or equal to OP1.

**RETG** Return to mark greater than or equal to OP1. RET assumes the mark to be 1. The given mark in the operand must be a constant.
SKIP  No action is taken. The value of a skip is undefined.

PROC  Routine denotation. The first operand is a pointer to the serial clause or unit to be compiled as a routine. All current information is stacked and the code generator is reset to its initial state before the routine is compiled.

REF   The REF operator. The operand is elaborated in reference mode.

VAL   The VAL operator. The operand is elaborated in the current mode and the resulting address word is fetched through.

LOCAL LOCAL declaration. The first operand is a symbol table pointer. The second operand is the initialization unit. The priority of the identifier is checked to determine if it should be loaded to a register. If not it is placed on the stack.

COMMA COMMA node. This node is used to chain together parameters, indexers in FORALL nodes, CASES, and so on.

FOR   FOR indexer. See DO node. The first operand is the indexer identifier. The other operands are the initial value, step size, and limit, respec-
tively.

FORALL indexer. The first operand describes a list of indexers separated by COMMA nodes. The second operand is similar list of initialization units.
REFERENCES


