A MACHINE INDEPENDENT APPROACH TO AUTOMATIC CODE GENERATION

by

André Scheunemann

A thesis submitted in conformity
with the requirements for the
degree of Master of Science
in the School of Computer Science
McGill University
June 1982 ©

Abstract

There has been extensive research into the automatic generation of compilers. This research has largely automated the generation of the analysis phase. Less progress has been made, however, on the less formal code generation phase. This thesis presents a table driven approach to the automatic generation of code. An automatic method for selecting and joining code templates so as to produce near-optimal code has been developed. The basic approach is quite independent of the target machine architecture. Retargeting the code generator for a new machine requires little more than providing new tables for the algorithm. This approach is very practical as a complete code generator for the Pascal language has been implemented and is currently used for the IBM 370. To demonstrate the applicability of our method to different machines, additional implementations for the PDP-11 and the VAX-11 are also discussed.

Abrégé

Il y a eu d'amples recherches sur la génération automatique des compilateurs. Cette recherche a largement automatisé la génération de la phase analytique. Moins de progrès ont été faits cependant, sur la phase moins formelle de la génération de code. Cette thèse présente une approche à la génération automatique de code, fonctionnant à l'aide de tables. On a développé une méthode automatique pour sélectionner et joindre des séquences de code individuelle de façon à produire du code quasi optimal. L'approche de base est tout à fait indépendante de l'architecture de la machine pour laquelle le code doit être généré. Changer le générateur de code de façon à produire du code pour une nouvelle machine, ne requiert que le changement des tables utilisées par l'algorithme. Notre approche est très pratique car un générateur de code pour le language Pascal a été implanté et est présentement utilisé sur le IBM 370. Pour démontrer l'adaptation de notre méthode à d'autres machines, des implantations additionnelles sont aussi présentées pour le PDP-11 et le VAX-11.

Acknowledgements

I wish to express my gratitude to all the people who helped make this thesis a reality. The faculty members of McGill University who provided the academic environment and background required for such an endeavor. My friends, relatives and fellow graduate students who provided encouragement and help. I am especially grateful to my advisor, Professor Nigel Horspool, for his invaluable guidance, criticisms, technical discussions and many insights into the problem which this thesis treats. I would also like to thank my parents for encouraging me in my academic persuits.

The financial support of the Natural Sciences and Engineering Research Council of Canada is also appreciated.

TABLE OF CONTENTS

FIGURES

TABLES

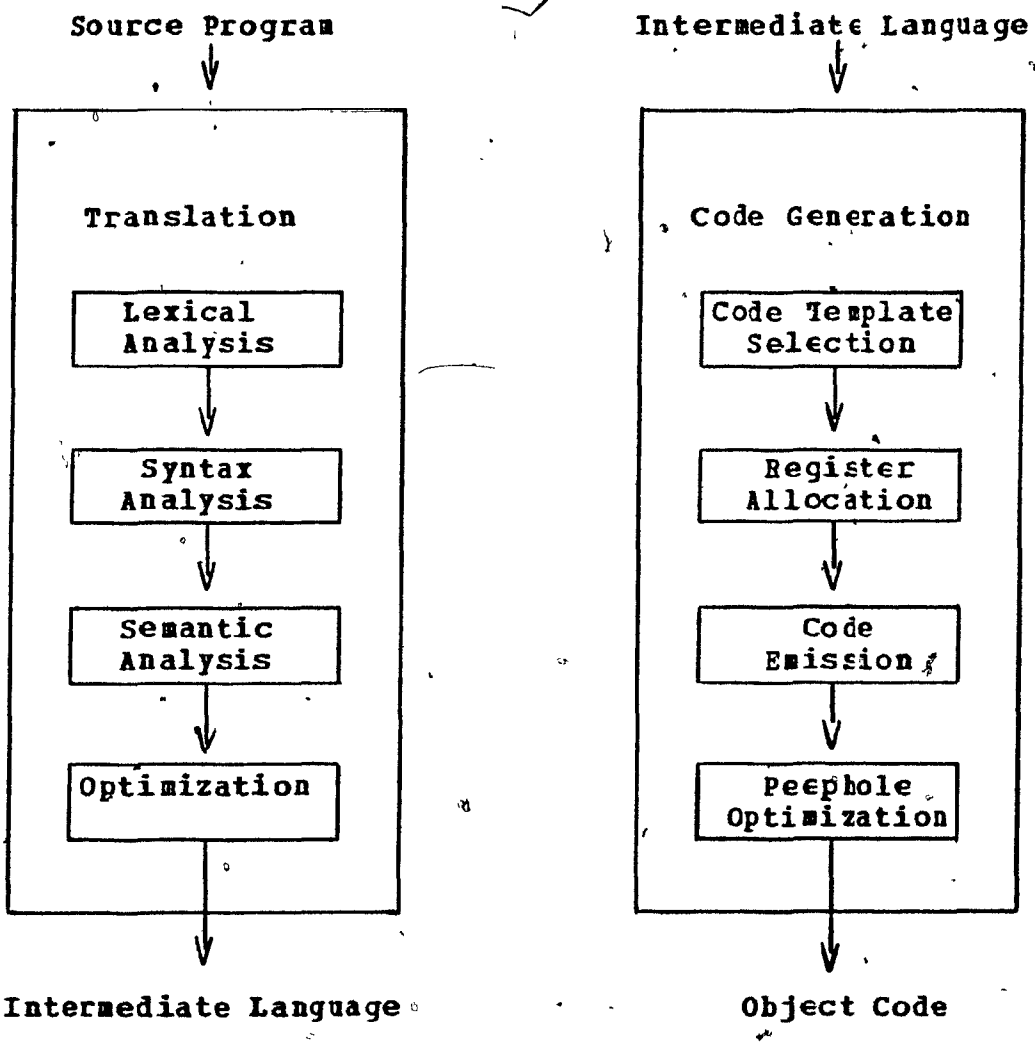
Chapter 1

Introduction and Previous Research

1.1 Overview of the Compilation Process

A compiler is a program that translates other programs written in a high-level language into executable code for a specific target machine [1,2,3]. The translation process is often performed as two separate major phases. The first phase is that of translating the program written in a particular high-level language into an intermediate language. This phase is further subdivided into several modules each performing a specific task. These modules include: Lexical Analysis, Syntax Analysis, Semantic Analysis and optionally an Optimization module. A schematic diagram of this phase of a compiler can be found in Figure 1.1a. The second phase, which is called code generation, then converts the intermediate language generated by the first phase into machine code for a specific target computer. This phase includes operations such

as Code Template Selection, Register Allocation, Code Emission and probably Peephole Optimization [4,5]. Figure 1.1b gives a schematic view of the code generation phase of a compiler.



a) Translation Phase b) Code Generation Phase

Figure 1.1. Logical Compilation Phases and their Names.

1.2 Previous Research

Most research done on compilers in the past has been concentrated on the translation phase and this process is now well understood. Several tools exist to make the process of writing these translation modules more automatic. It is also possible for the first phase of a compiler to be largely machine independent. The translation phase of such a compiler needs little or no change when the compiler is retargeted to a different machine. This is achieved primarily by using a machine-independent intermediate language. It is only in the last few years, however, that much research has been performed on code generation itself, and it is not yet clear which is the best approach to the problem.

1.2.1 Theoretical Code Generation

Past research in code generation can be divided into three main categories. There was research that dealt with code generation from a theoretical viewpoint [6,7,8] and was mainly oriented towards the optimality of the code generated. Usually, only hypothetical, well-behaved, instruction sets were considered (i.e. instruction sets where all registers are equivalent, where all addressing modes can be used with every instruction and where there are no specialized instructions that correspond to exceptional cases of other instructions). The methods developed by these people did not, in most cases,

take into account the problems involved with retargeting the code generator.

1.2.2 Procedural Code Generation

The other two categories of research were more practical approaches to code generation and dealt with existing computer architectures. The second class of research involved methods of providing information about a target computer in a procedural way using special purpose code generation languages and interpreters. This approach was taken by Elson and Rake [9] who developed a high-level definition language which was used to define macros for each possible node that could appear in the syntax tree, produced by the parser, representing the compiled program. Each of these macros included all the logic that was needed for optimization, error detection and code generation. An interpreter would then be used to execute the macros as the corresponding nodes occurred in the program tree. The generated code was optimized, as each special case in the language could be separately handled in these macros. The quality of the code, however, was obtained at the expense of transportability since each macro had now become much more machine dependent. This method also has the disadvantage that all these code generation macros must be written by the compiler implementor and thus strongly depend on his ability to design and debug them.

A similar approach was also taken by Wilcox [10]. The analysis phase of his compiler first constructed a tree representation of the program. This tree was then transformed, using a translator, into a sequence of instructions for a hypothetical machine based on the operations of the source language; the machine was called Source Language Machine or SLM. The choice of SLM instructions was made so as to facilitate optimization and transportability. The way in which the program tree was translated into SLM instructions was determined by a mapping which ultimately transformed each operator node of the tree into a sequence of SLM instructions. The next phase, called the coder, would then translate the SLM representation of the program into object code for a specific machine. For each SLM instruction, the compiler writer provided a routine, written in a special coding language (similar to an assembler language), specifically designed for a particular object machine. To retarget the compiler, this special coding language would have to be totally redesigned and then, using this new language, all the code generation routines rewritten. The work involved is clearly no small task and once again depends heavily on the implementor's ability to design and debug the code generation routines.

The approach taken by Donegan [11] was again very similar in that it used a code generation preprocessor language, CGPL. The routines used to determine which code sequences were to be

generated for the nodes in the expression trees, would then be translated, by the CGPL, into PL/1. As in the case of Wilcox [10], these routines had to be written for each different operator that could appear in the trees, but unlike the special language he used for this purpose, the CGPL used by Donegan was a high-level language which made the process of writing these code generation routines much simpler. This method is different from the others as the operand storage locations (register, memory, condition code) for a given operator, determined the state in which the coder was in. This feature made the code generation process a little more automatic. The quality of the code generated was similar to that of the code produced by Wilcox's method. One of the major problems with Donegan's method as well as with the others, is that operators were translated individually after all operands had been evaluated and without regard to the requirements of subsequent operators.

While code generation languages are great improvements over the method of writing a code generator in a normal high level language, they still leave us with several problems, particularly when retargeting the code generator is a major concern. In addition, these languages still leave most of the low level decisions up to the implementor. Certain improvements over these methods would be desirable. First of all, retargeting the code generator should be simpler, thus reducing the amount of work that must be done by the

implementor and at the same time probably increasing the reliability and quality of the code generator. In a sense, the code generation process should be made more automatic. Ideally, one wishes only to specify the characteristics of the machine, and have a system which automatically produces a code generator tailored to the specific machine. In addition, it is important to maintain a high standard in the quality of the generated code.

1.2.3 Code Generation by Semantic Description

The third class of research, was an attempt to provide the target machine information in a table format, data base or other descriptive form. This information was analyzed, possibly transformed and then used in the code generation process. The method presented here would fit into this last category. Some of the first work done in this area was by Miller [12]. His approach consisted in converting the program to be compiled into a sequence of machine independent macros. He then would provide a description of the target machine in which all operators that appeared in the macros were associated with a sequence of instructions to implement each operator on the target machine. Automatic conversion from one storage type to another was provided for cases where certain macros could only be applied to operands of specific storage types. A major problem with Miller's work is that many simplifying assumptions were made about the target machine architectures.

Weingart's [13] approach was quite different from Miller's in that the compiled program was translated into a low-level intermediate language. This language was close enough to the target machine language to make code generation possible simply by comparing partial parse trees with the target machine instructions, which were also represented as parse trees in the intermediate language. Automatic conversions from storage to register and vice-versa were provided when these actions would allow a match between part of the tree and an instruction. One disadvantage with Weingart's approach is that the intermediate language is too heavily dependent on the target machine's instruction set. Thus even machines with small differences in instruction sets or addressing modes may require that the intermediate language be changed. Also the expression trees must be optimized to ensure high quality code as well as efficiency of the code generation algorithm. The main advantage is that his method of describing an instruction set is easier to use and less error prone than other methods which involved writing code generation routines or macros.

The work done by Newcomer [14] involved a machine descriptive language, MDL, which was specifically designed for the generation of machine code for a given language. Code templates were produced from a machine description in MDL. Unlike other approaches, one of Newcomer's major goals was to produce highly optimized code. His code generator was heavily

influenced by previous work done on the ELISS compiler [15,16]. His code generator functioned in a similar way to Weingart's [13]. It searched for templates that would evaluate expression trees produced by a previous phase of the compiler. However, unlike Weingart's method which found any possible code sequence for an expression tree, Newcomers's method would find the best code sequence (according to some criteria such as space, execution time or some combination of both) for the entire tree. Needless to say, the quality of the generated code was quite superior to what had been previously achieved using this type of code generator. On the other hand, a lot of extra work was done trying to optimize the program before the actual code generation phase. The most important problem with Newcomer's method is probably that the search technique used to find the optimal code sequence for an expression tree is very simple as it simply enumerates all possible code sequences and then simply picks the best one. This approach is not as efficient as it should be.

This brings us to the work done by Fraser [17]. It essentially involved a system called XGEN that produced code by analyzing a machine description. The description was provided using a special purpose language called ISP. It not only described the instruction set but also all possible storage locations including memory, registers, condition codes etc. First of all, the compiler transformed the program into an intermediate language called XL. XGEN would then expand

these XL instructions using machine independent macros, rewriting them in terms of the ISP of the target machine. Subsequently, using a specific set of rules, it would generate the actual assembly code for the specific target machine. Using this system, the generation of code for a different machine would require that a description of the new machine be given in terms of the ISP and possibly also require the addition of new rules to the XGEN program.

Glanville's approach [18] was similar to that of Weingart [13]. First a program was translated into a very low-level machine independent intermediate code. Then a special parser would convert the intermediate code into the target machine's code using standard SLR(1)[1] parse tables and a series of production rules that described the target machine's instruction set. This code generation method would choose the best available machine instructions to execute a given sequence of intermediate language instructions. Specialized machine instructions were used whenever possible. Several algorithms were developed for building the parsing tables, detecting possible loops and for guaranteeing that correct

---

[1] Although the SLR(1) parse tables are standard, the table construction algorithm will accept any context free grammar, not necessarily unambiguous. In the general case the language accepted by the resulting parser is not guaranteed to be the same language defined by the grammar. If the grammar satisfies certain sufficient conditions, then the tables are built in such a way that the language accepted by the parser is the same as that defined by the grammar.

code would be generated. Retargeting was easily achieved by changing the production rules to reflect the instruction set of the new machine. From this description the SLR(1) parse tables are generated without any intervention from the implementor. Of all the methods discussed so far this is surely the most powerful and also the easiest to retarget. One of the nicest features of Glanville's method is that the code generated is guaranteed to be correct as long as the machine description accurately conveys the computer's architecture.

Cattell [19], in his work on a production quality compiler, developed a method by which he would generate a set of tables from a machine description and an intermediate language called TCOL. These tables contained a mapping from TCOL operators to machine operators. Once these tables were created, the code generator simply traversed the program tree, expressed in terms of the TCOL operators, comparing parts of it against patterns on the left hand sides of the productions in the tables. When a pattern was found to match, the right hand side of the production would specify the code to be generated as well as special compiler actions (such as storage allocation) and further matches to be recursively performed. Cattell's method is also very automated as retargeting simply requires the new machine description and the generation of new tables.

Code generation from a semantic machine description is a promising area in compiler research. Its main goal is to free the compiler writer from having to analyse numerous special cases when choosing code sequences and allows the entire code generation process to be viewed from a higher, more abstract, level. In this way, more of the implementor's time can be spent studying the problem rather than searching for possible solutions. The fact that a code generator can be automatically created provides the possibility of writing a practical, easily retargetable compiler, that will not require years of development time. This is becoming increasingly important with the growing number of computers that are available today.

As is apparent from the past research in this area, there are two ways of generating code from the semantic description of an instruction set. One way is to have an intermediate language that is fairly high-level and to obtain code sequences for each operator in this intermediate language in terms of the target machine's instruction set. These code sequences are then emitted for each corresponding intermediate language operator that is encountered. The other way, is to use a very low-level intermediate language and then to express the target machine's instruction set in terms of this intermediate language. A process similar to pattern matching is then used to map target machine instructions onto the intermediate code.

1.3 This Work

The approach presented in this paper is similar to the former one in that the intermediate language is not low-level. This approach is well suited for generating high quality code, and at the same time, facilitates retargeting of the code generator. An automatic method for selecting and joining code templates so as to produce near-optimal code has been developed. The basic approach is quite independent of the target machine architecture. Retargeting the compiler for a new machine requires little more than providing new tables for the algorithm. As Wulf [20] points out, a code generator must essentially enumerate all the plausible code sequences and pick the most suitable. The method presented here is certainly enumerative, but several techniques have been incorporated to keep the enumeration under control.

Chapter 2

Code Generation Principles

Intermediate code in a compiler can take many forms, such as triples, indirect triples, quadruples or code for a hypothetical stack machine [3]. Let us assume that the intermediate code has the form of a generalized expression tree. The nodes of the tree correspond to operations of a stack computer. As an ongoing example, consider the Pascal statement:

if ((A+20) <= (B*C)) and (FLAG or (D>0)) then goto 10

which could be translated to the intermediate code shown in Figure 2.1. The operations that appear in the tree are (or are similar to) Pascal P-code instructions [21]. The meanings of the p-operations used in the figure are summarized in Appendix A. P-code operations represent instructions for a hypothetical stack machine called the P-machine. Each P-instruction can be classified into one of three groups. First there are instructions that do not operate on any data.

For example, LDC and LOD represent such P-instructions. They are mainly used in setting up data for subsequent instructions. When executed in the P-machine, they push data onto the stack. If P-instructions are structured into an expression tree, they represent the leaf nodes. Then there are single operand instructions such as NOT and FJP. When executed, they take their operand from the top of the stack. In a tree representation, the operand corresponds to the sub-tree of that node (FJP or NOT). Finally, there are instructions, such as ADI, EQU and OR that operate on two operands located on top of the stack. In the tree representation, they are the nodes with two sub-trees. Since other intermediate code forms are easily convertible to expression tree form, there is little or no loss of generality in considering expression trees.

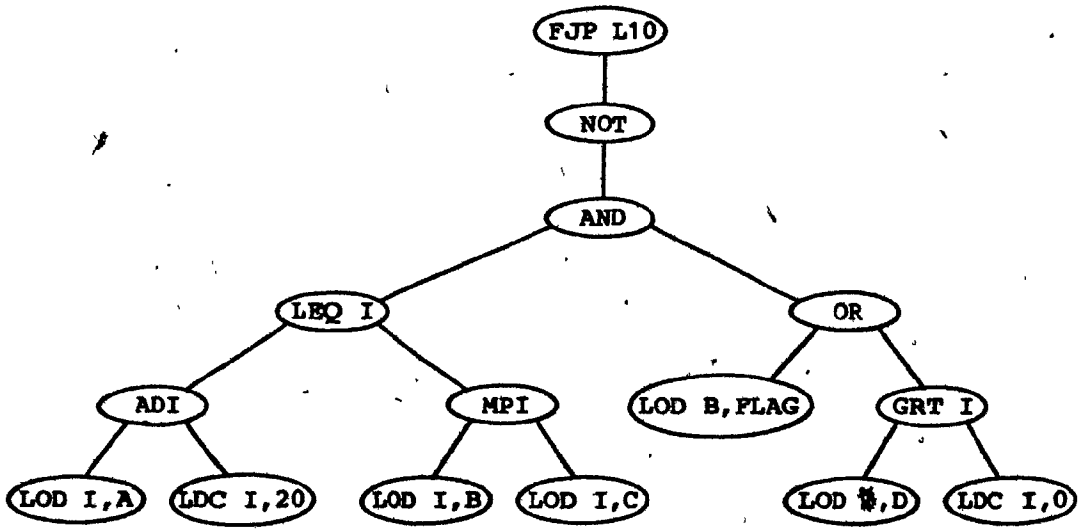


Figure 2.1. Sample Expression Tree for a Pascal Statement.

Before considering translation of the intermediate code into machine code, it should be pointed out that various machine-independent optimizations can be applied to the expression trees. Some obvious optimizations, such as folding of constant expressions, should definitely be applied because they also tend to reduce the complexity of the subsequent code generation process. Some other optimizations, such as finding common subexpressions, make the code generation process more complex. (Common subexpression elimination may destroy the tree structure, transforming it into a DAG.) It is also useful to break up certain P-instructions into others that already exist, but that are simpler or more general. In doing so, one reduces the number of P-instructions that must be considered by the code generator, thus reducing its size and complexity.

When the intermediate code is translated, the expression tree gets mapped into a corresponding computation on the target computer. The mapping process to perform this translation can be very complicated but, in its simplest form, each subexpression in the tree is translated into code that will generate the value of that subexpression. The place where that value will reside in the target computer depends on the possibilities offered by the machine architecture and on the selections made by the code generator. For a typical computer, the choices include: memory, register, condition-code register (which often represents a Boolean

value), and program counter (which can implicitly encode information, as will be seen later). Once these choices are made, the code generation process is quite tightly constrained.

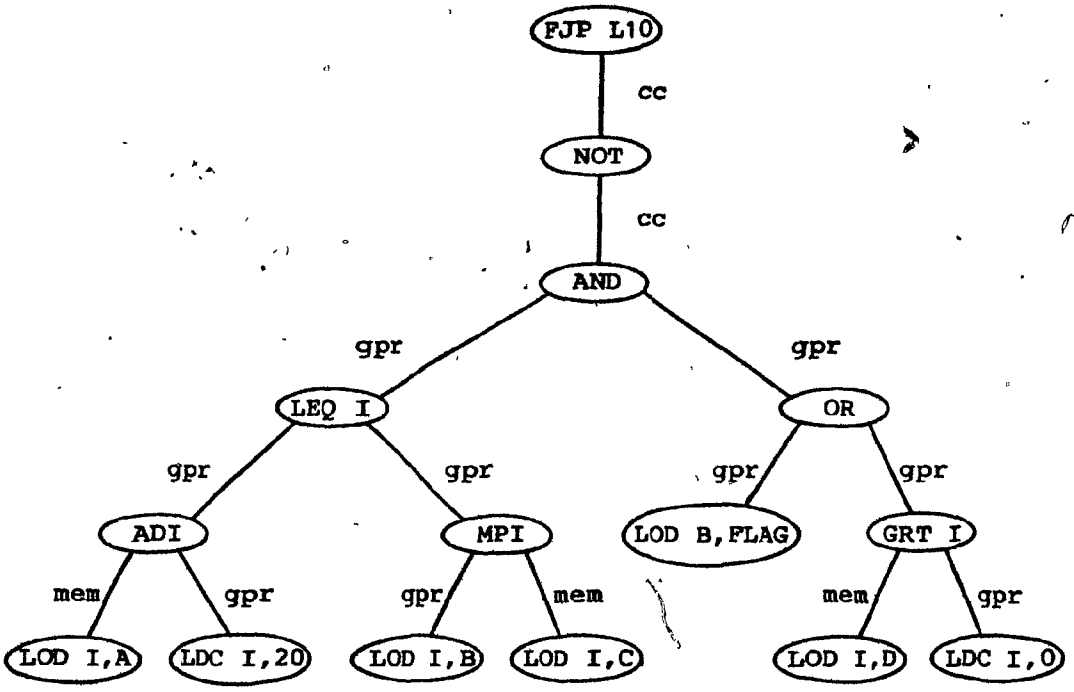


Figure 2.2. Labelled Expression Tree.

Let us now return to the tree of Figure 2.1. A plausible labelling for the IBM 360/370 series of computers that shows where the values of subexpressions could reside, is given in Figure 2.2. (This is not an optimal labelling.) Given this particular labelling, the code generator is constrained to produce code along the following lines:

```
LA   R1,20      {Result of LDC I,20   put into gpr 1}
A    R1,A       {Result of ADI        put into gpr 1}
L    R2,B       {Result of LOD I,B    put into gpr 2}
M    R2,C       {Result of MPI      created in gpr 3}
CR   R1,R3      {Result of LEQ temporarily in the
LA   R1,1        condition code, but is converted
BLE  *+6         to a 0 / 1 value in gpr 1        }
SR   R1,R1
SR   R2,R2      {Result of LOD B,FLAG put into gpr 2}
IC   R2,FLAG
SR   R3,R3      {Result of LDC I,0    put into gpr 3}
C    R3,D       {Result of GRT temporarily in the
LA   R3,1        condition-code, but is converted
BLT  *+6         to a 0 / 1 value in gpr 3        }
SR   R3,R3
OR   R2,R3      {Result of OR is  0/1 value in gpr 2}
NR   R1,R2      {Result of AND is in condition code }
BNZ  L10        {Effect of NOT and FJP L10 combined }
```

The code generation process can thus be decomposed into two main steps:

1. Label the expression tree with storage classes to be used for holding values of the various subexpressions. The labelling should correspond to an efficient code pattern for the target computer.

2. Traverse the tree, selecting code sequences (or "code templates") that accept operands and generate results in the specified storage classes.

In principle, code templates for every P-code operation using every meaningful combination of storage classes could be provided. However, the semantics of P-code and the permitted usage of some machine resources in the target computer make many combinations meaningless. Such templates, therefore, are not provided. Each template can be associated with a cost.

This cost might, for example, be the estimated execution time for the code sequence, or the length of the code sequence, or some combination of the two. The cost of a non-existent template is implicitly infinite.

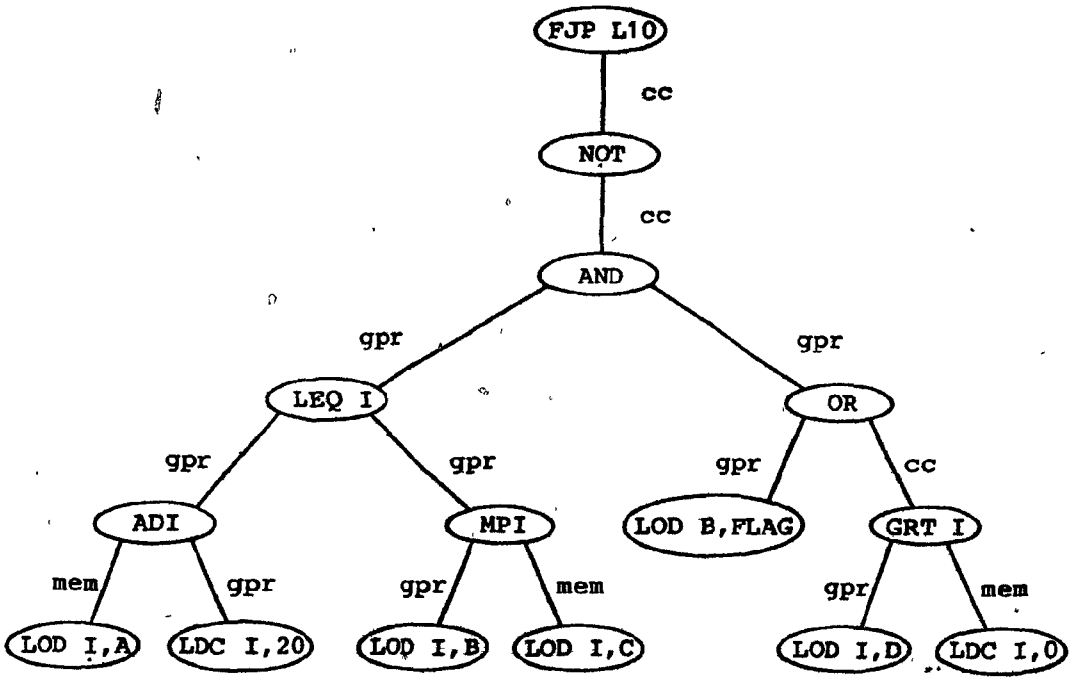


Figure 2.3. A Better Tree Labelling.

In theory, all meaningful labellings of the expression tree could be enumerated and then the labelling that has the lowest total cost selected. Producing machine code that conforms to this labelling will give us optimal code for the entire expression tree. As an example, consider the labelling for the tree of Figure 2.3. The tree is the same as the one of Figure 2.2 but the labelling is different. With this

labelling certain features of the IBM 360/370 instruction set (such as the use of condition codes set by arithmetic instructions) are taken advantage of to produce the following code:

```
LA   R1,20      {Result of LDC I,20   put into gpr 1}
A    R1,A       {Result of ADI        put into gpr 1}
L    R2,B       {Result of LOD I,B    put into gpr 2}
M    R2,C       {Result of MPI      created in gpr 3}
CR   R1,R3      {Result of LEQ temporarily in the
LA   R1,1        condition code, but is converted
BLE  *+6         to a 0 / 1 value in gpr 1          }
SR   R1,R1
SR   R2,R2      {Result of LOD B,FLAG put into gpr 2}
IC   R2,FLAG
L    R3,D       {Result of LOD I,C    put into gpr 3}
LTR  R3,R3      {Result of GRT temporarily in the
BLE  *+6         condition code                      }
LA   R2,1       {Set true value in gpr 2            }
NR   R1,R2      {Result of AND is in condition code }
BNZ  L10        {Effect of NOT and FJP L10 combined }
```

The code sequence above is 2 instructions shorter than the one for the tree of Figure 2.2. Using the number of instructions per code sequence as a criterion, one can say that the labelling of Figure 2.3 is better than the one of Figure 2.2. In fact this code sequence is the best possible one that can be achieved using the storage classes discussed so far. A better code sequence is still possible using an additional storage class as will be demonstrated further on in this chapter. Note that the labelling given in Figure 2.3 is only valid for a left to right evaluation of the expression tree as the condition code can only hold a value temporarily. Since many machine instructions will alter the condition code, it can only be used in a restricted manner. It may not always be possible to precisely determine in advance the cost of a

template. For this reason, optimal code generation will not always be possible. However, in most cases, the machine code generated will be the best possible without performing sophisticated optimizations.

Of course, a storage class label does not provide complete information by itself. Along with the label, we should associate additional information such as the number of a register or the address of a memory location to be used. However, this information can usually be added after the labelling of the tree has been performed and after all code templates have been selected.

The concept of a storage class can be used in a very general way. For example, short-circuit evaluation of Boolean expressions [1] can be handled by adding a SCB ("Short-Circuit Boolean") storage class. Short-circuit evaluation is sometimes referred to as "McCarthy Evaluation" (from the COND function in LISP). Treating SCB as a storage class in the same way as the other storage classes, will give us the labelling of the sample expression tree that is shown in Figure 2.4. This tree would then be translated to IBM 360/370 code similar to the following:

```
        LA   R1,20                      BNZ  L10
        A    R1,A                       SR   R1,R1
        L    R2,B                       C    R1,D
        M    R2,C                       BLT  L10
        CR   R1,R2              FALSE   .
        BH   FALSE                      .
        TM   X'01',FLAG                 .
```

The boolean value corresponding to a SCB storage class is actually encoded in the program counter when the code is executed. However, this does not prevent us from treating SCB as a storage class. Associated with the SCB label, information about the "true" and "false" exits out of a block of code (but only when the code is actually generated) would be needed. A code template for a P-code operation such as "AND" with SCB operands would merely require book-keeping work on the information and would not usually require any machine instructions to be generated.

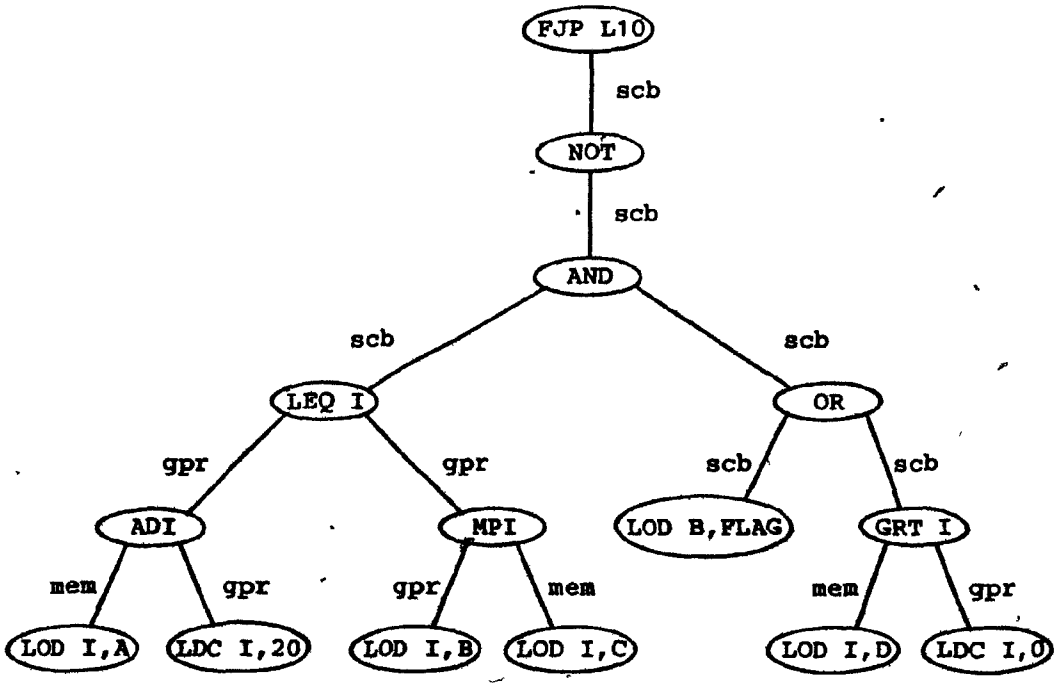


Figure 2.4. Labelled Tree with Short-Circuit Booleans.

Chapter 3

Practical Implementation for the IBM 360/370

The basic method of Chapter 2 has two main drawbacks. First, the number of code templates would be excessive. Second, the storage classes need further refinement if high quality code is wanted. In this chapter we will now show how the method can be evolved to solve both problems.

The number of templates can be greatly reduced by a factorization process. Consider, for example, the P-code operation ADI to add two integers. A suitable template for ADI when both operands and the result have the "memory" storage class and when the target machine is the IBM 360/370 is:

```
L   R1,A    {Load first operand into register }
A   R1,B    {Add the second operand           }
ST  R1,T    {Store result into temporary memory}
```

However, this code is similar to the template needed for adding a "register" operand to a "memory" operand leaving the

result in a "register" operand. The difference is that the template is surrounded by load and store instructions. The load essentially converts from the "register" class to the "memory" class and the store implements the opposite conversion.

By having implicit, automatic, conversions between storage classes whenever convenient, the number of elementary templates needed for the ADI operation is reduced to approximately the number of suitable machine instructions for integer addition. With the IBM 360/370 architecture one might need only templates corresponding to the AR (add register) and A (add memory to register) instructions. All of the P-code operations are amenable to this factorization into basic templates plus conversions. All that is needed is an additional group of templates which are used only for performing conversions from one storage class to another.

One also notes that many P-code operations, such as ADI, are commutative and the encoding scheme for templates can take advantage of this to reduce the storage occupied by template tables even further.

To demonstrate how refinement of the storage classes is useful for generating good code, a practical example will be provided. The example deals with the IBM 360/370 series of computers [22]. With this architecture, most instructions that access memory (RX format instructions) use an address composed from a base register, an index register and a fixed

displacement. However, many other instructions (RS and SS format instructions) access memory with an address composed from only a base register and a displacement. If the code generator can discriminate between the two address formats, it will be steered towards using the most appropriate instruction sequences. Thus memory storage classes XBD and BD are introduced to correspond to these two cases. In addition, a distinction must be made between a memory location and the address of that location. Thus, two new storage classes are introduced to correspond to a memory address. Since they are of the same format as a memory location, they are referred to as the XBDA and BDA storage classes. The difference between the XBD and XBDA storage classes is that the XBD class refers to a memory location that holds a value which is directly accessible and which can be used as an operand for all RX format instructions while the XBDA class represents an address in memory where data objects reside. In the case of the XBD class, the data item that is referenced can have a maximum length of 8 bytes, as this is the size of the largest data type that can be directly used as an operand in an RX format instruction, while there is no restriction on the length of the data object referred to using the XBDA class as any operation performed on this object will be carried out using its address. As an example, the value of a variable in memory would have the XBD storage class while the base address of an array used as an operand in the calculation of an element address would be referenced with the XBDA class. Further

storage class refinements can be applied to improve on the quality of the code that can be produced. One such refinement could be the subdivision of the "register" class into three sub-classes referred to as the "even register" class, the "odd register" class and the "double register" class. This subdivision will permit better usage of the multiply and divide instructions as well as the registers themselves (since unnecessary loads between even and odd registers will be avoided). Considering the heavy usage of the multiply instruction in typical programs, this simple refinement can improve the overall quality of the generated code significantly. A list of suitable storage classes and a summary of possible conversion templates is given in Table 3.1. Note that many other conversions can be synthesized by concatenating two or more basic conversion templates. It should also be noted that the list of storage classes provided here can be further refined to take advantage of other features of the instruction set. In general, the more storage classes that are used, the better the quality of the code that can be produced (but at the expense of requiring more templates, more conversions, and much larger tables and therefore a slower algorithm for selecting the right templates).

Recommended Storage Classes for IBM 360/370

GPR	General Purpose Register
EVR	Even Register
ODR	Odd Register
DOR	Double Register (Even-Odd pair)
FPR	Floating Point Register
XBD	Memory (Index/Base/Displacement)
BD	Memory (Base/Displacement)
XBDA	Address Constant (XBD format)
BDA	Address Constant (BD format)
CC	Condition Code
SCB	Short-Circuit Boolean

Elementary	GPR	-->	EVR	{ No code generated }
	GPR	-->	ODR	{ No code generated }
	EVR	-->	GPR	{ No code generated }
	ODR	-->	GPR	{ No code generated }
	DOR	-->	ODR	{ No code generated }
	GPR	-->	BDA	{ No code generated }
	BD	-->	XBD	{ No code generated }
	BDA	-->	XBDA	{ No code generated }
Conversion	XBD	-->	GPR	{ L instruction }
	XBDA	-->	GPR	{ LA instruction }
	CC	-->	GPR	{ LA,BC,SR sequence }
	SCB	-->	GPR	{ LA,BC,SR sequence }
	ODR	-->	EVR	{ LR instruction }
	EVR	-->	ODR	{ LR instruction }
Templates	EVR	-->	DOR	{ SRDA instruction }
	ODR	-->	DOR	{ LR,SRDA sequence }
	XBD	-->	FPR	{ LD instruction }
	XBD	-->	BD	{ AR instruction }
	GPR	-->	CC	{ LTR instruction }
	BD	-->	CC	{ TM instruction }
	CC	-->	SCB	{ BC instruction }

Table 3.1. IBM 360/370 Storage Classes and Elementary Conversions.

A sample expression tree labelled with IBM 360/370 storage classes is given in Figure 3.1. The arcs between the nodes of the tree are each labelled with two storage classes. One of them represents the result storage class of the sub-tree below and the other represents the storage class required as the operand for the next operator node. A storage

conversion is required whenever these two storage classes are different. Figure 3.2 shows the same expression tree with the addition of explicit conversion operator nodes. Note that some of these conversions are free and do not require any code to be generated while others are not necessarily elementary conversions, as can be seen from Table 3.1, and that they may require the expansion of several code templates.

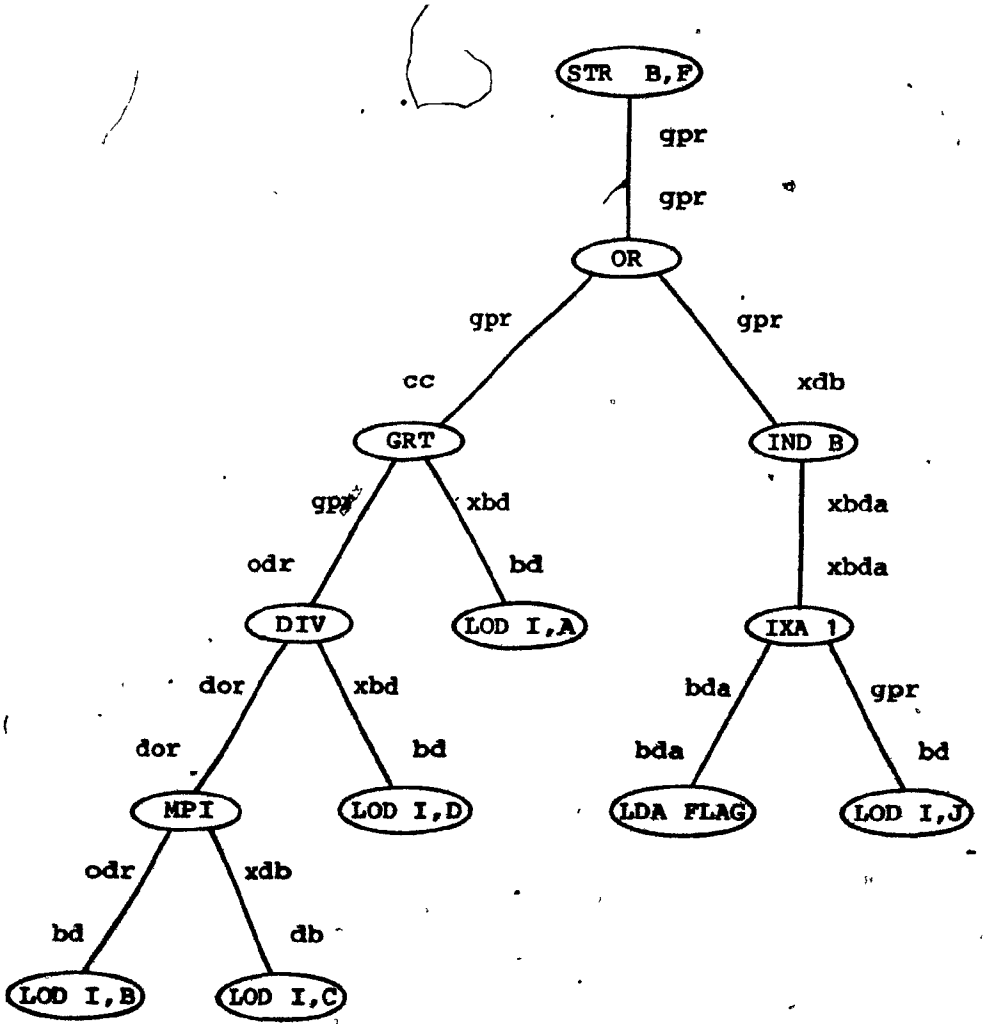


Figure 3.1. Expression Tree Labelled with IBM 360/370 Storage Classes.

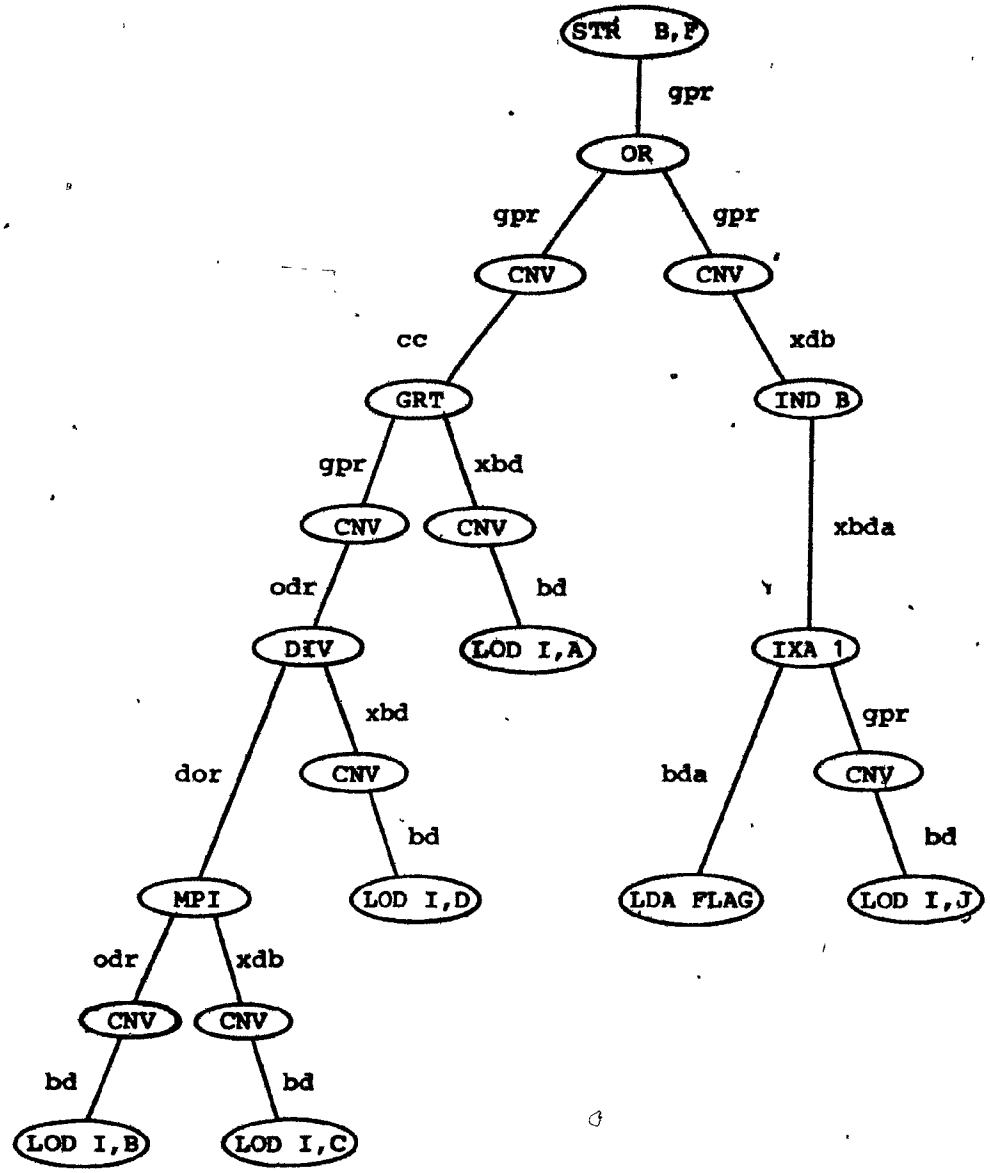


Figure 3.2. Labelled Expression Tree with Explicit Conversion Operators.

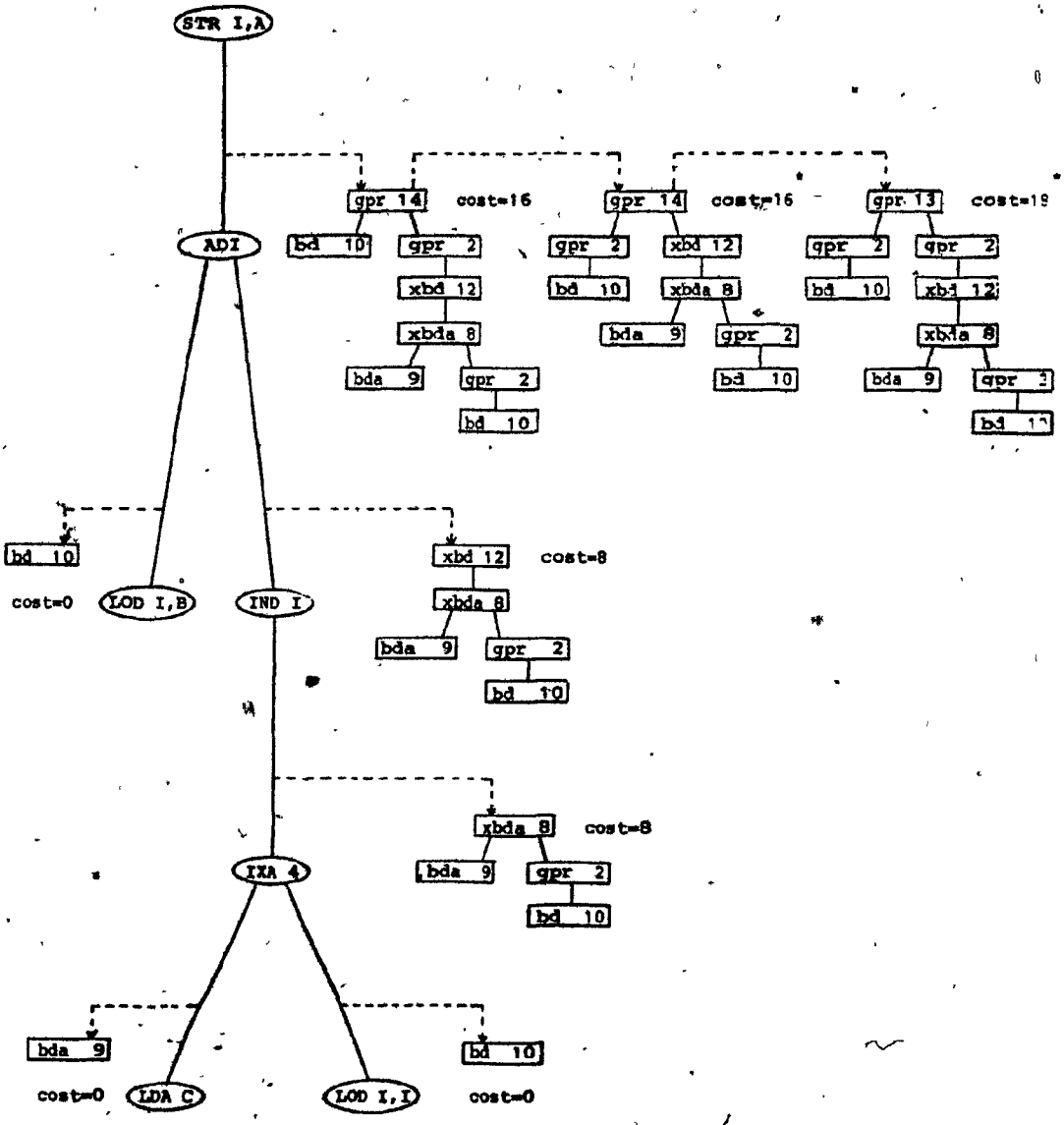
The process of finding the best sequence of code templates (i.e. best tree labelling) for a given expression tree is handled by an algorithm that essentially performs an exhaustive search for the best sequence over all plausible

template sequences. The complete algorithm is detailed, using a mixture of English and Pascal, as follows:

```
procedure BUILD_SOLUTION_LIST( EXPRESSION_TREE, RESULT );

begin
if EXPRESSION_TREE is leaf_node then
    RESULT := list of single node trees where each node is a template
              that implements this (elementary) expression

else begin
  RESULT := (empty list);
  BUILD_SOLUTION_LIST( LEFT  sub-tree of EXPRESSION_TREE, LEFT_LIST );
  BUILD_SOLUTION_LIST( RIGHT sub-tree of EXPRESSION_TREE, RIGHT_LIST );

  for  TEMPLATE := each template for the current P-code instruction  do
    for  LT := each tree in LEFT_LIST  do
      for  RT := each tree in RIGHT_LIST  do
        if  there exist conversions from the resulting storage classes
            of LT and RT to the storage classes required as operands
            by TEMPLATE  then
          begin
            STOR_CLASS :=  resulting storage class of TEMPLATE;
            COST :=  cost of using TEMPLATE plus costs of evaluating
                     and converting operand sub-trees;

            for  T := each tree in RESULT  do
              if  cost of T + cost of conversion from result storage
                  class of T to STOR_CLASS  <=  COST  then
                goto EXIT_LOOP;

            for  T := each tree in RESULT do
              if  conversion cost from STOR_CLASS to resulting storage
                  class of T  <=  cost of T  then
                remove T from RESULT;

            concatenate the tree formed with TEMPLATE as its root
              and LT, RT as operand sub-trees onto the end of RESULT;
EXIT_LOOP: end
end
end    BUILD_SOLUTION_LIST
```

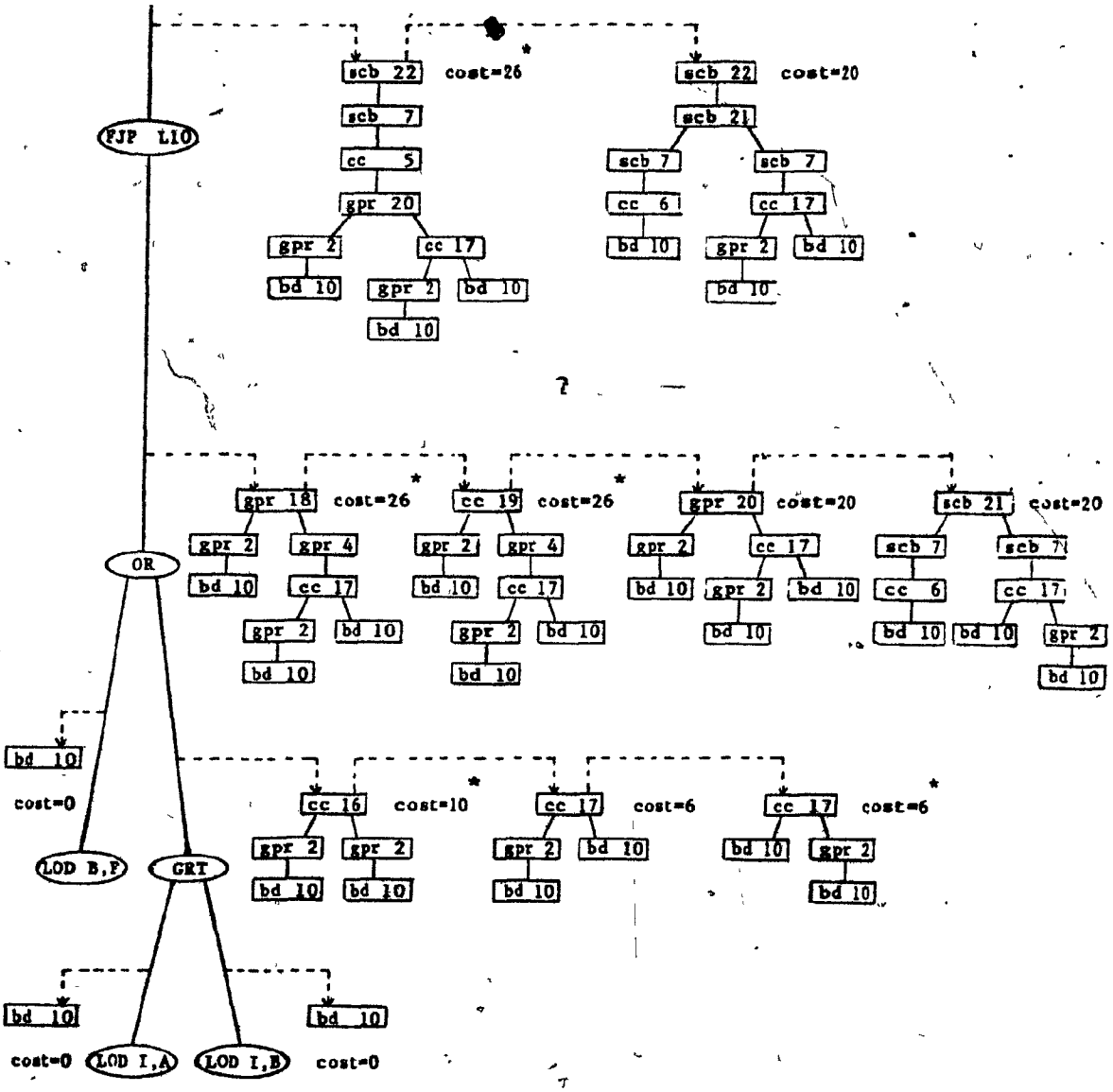
Figures 3.3 and 3.4 show an expression tree along with their solution trees as they would be generated by the algorithm.

The rectangles in the figures hold the following information: <storage class ; cost table entry # >.



* Indicates solution trees that are dropped because better or equally good solutions have been previously generated.

Figure 3.3. Sample Operation of the Code Template Selection Algorithm.



* indicates solution trees that are dropped because better or equally good solutions have been previously generated.

Figure 3.4. Sample Operation of the Code Template Selection Algorithm.

The algorithm relies heavily on the use of two tables. The first table holds the information about the use of all possible templates for all P-code operations. Each entry in this table holds the information described in Table 3.2. The cost table entries required by the algorithm for the expression trees of Figures 3.3 and 3.4 are given in Table 3.3. In this table, the left and right storage classes appearing in the RESULT column refer to the storage classes of the left and right operands respectively. The "--" notation indicates an item of information that is not required by the table entry. The TEST field is used in the case of the CNV operation to distinguish between the load of an integer and a boolean. The template numbers refer to the following code sequences:

```
Template  0 :   No code
Template  1 :   LA    R,D(X,B)
Template  2 :   L     R,D(X,B)
Template  3 :   SR    R,R ;  IC  R,D(X,B)
Template  4 :   LA    R,1 ;  BC  mask,*+6 ;  SR  R,R
Template  5 :   LTR   R,R
Template  6 :   TM    D(B),X'01'
Template  7 :   BC    mask,??
Template  8 :   SLL   R,2
Template  9 :   AR    R,R
Template 10 :   A     R,D(X,B)
Template 11 :   ST    R,D(X,B)
Template 12 :   CR    R,R
Template 13 :   C     R,D(X,B)
Template 14 :   OR    R,R
Template 15 :   BC    mask,*+8 ;  LA  R,1
```

The second table that is used by the algorithm is the one that holds all the storage conversion costs and the conversion templates. A third table is required, but it is not directly used by the algorithm. This table comes into use in the final

code generation phase and holds the code sequences for each template. A schematic view of the entire code generation phase is given in Figure 3.5.

After the code templates have been selected (i.e. the best tree labelling has been determined), a register allocation algorithm traverses the expression tree and attaches register numbers to all the uses of a register storage class, such as GPR, EVR, ODR or DOR. If the expression tree makes use of the EVR, ODR or DOR storage classes (i.e. the IBM multiply or divide instructions are used) then the algorithm should replace all uses of the GPR storage class by either the EVR or ODR storage classes. This substitution is made in order to avoid subsequent unnecessary transfers between registers as a result of using an even register when an odd one would be more appropriate or vice-versa. If the semantics of the source language permit, the algorithm may rearrange the order of computation of expressions so as to minimize register usage. In the unlikely event that there are insufficient registers, the algorithm will modify the tree labelling by replacing a register storage class with a temporary memory class. The choice of which sub-expression to force into memory is determined by the "distance to next use" heuristic [23] in our implementation.

The actual code generation process has now become very simple. At this point, the nodes in the expression tree hold all the information (such as which template to generate and which registers are to be used in each template) needed to generate the code. This makes the code generation process little more than a simple tree traversal. It should be noted that the traversal does not force the postfix code generation property [1]. Our method of code generation provides the capability of generating code in a prefix or infix manner too. The ability to generate prefix code can be useful. Consider, for example, the following Pascal statement:

A := (B > D) or F

If one is constrained to use the postfix code generation approach, the code would normally be as follows:

```
L   R1,C                    SR   R1,R1
C   R1,D                    B    *+8
BH  TRUE            TRUE    LA   R1,1
TM  X'01',F                 ST   R1,A
BNZ TRUE
```

Using prefix code generation as well, the code sequence can be improved to:

```
LA  R1,1                    TM   X'01',F
L   R2,C                    BNZ  TRUE
C   R2,D                    SR   R1,R1
BH  TRUE            TRUE    ST   R1,A
```

As can be seen, the second instruction sequence generated is shorter by one instruction than the code generated in the strict post-order format.

OPERATION	:	P-code operation being implemented
LEFT_STOR_CLASS	:	Storage class of left operand
RIGHT_STOR_CLASS	:	Storage class of right operand
RESULT_STOR_CLASS	:	Storage class of result
COMMUTATIVE	:	Indicates commutativity of
	:	left / right operands
COST	:	Cost of using this template
TEMPLATE_NO	:	Index into table of templates
	:	where machine code is listed
SPECIAL_TEST	:	Indicates special tests for
	:	applicability are needed

Table 3.2. Information Contained in a Cost Table Entry.

ENT.#	OPER.	LEFT	RIGHT	RESULT	COMMUT.	COST	TEMP.#	TEST
1	CNV	xbda	---	gpr	no	4	1	--
2	CNV	xbd	---	gpr	no	4	2	integer
3	CNV	xbd	---	gpr	no	6	3	boolean
4	CNV	cc	---	gpr	no	10	4	--
5	CNV	gpr	---	cc	no	2	5	--
6	CNV	bd	---	cc	no	4	6	--
7	CNV	cc	---	scb	no	4	7	--
8	IXA	bda	gpr	xbda	no	4	8	--
9	LDA	---	---	bda	no	0	0	--
10	LOD	---	---	bd	no	0	0	--
11	IND	bda	---	bd	no	0	0	--
12	IND	xbda	---	xbd	no	0	0	--
13	ADI	gpr	gpr	left	no	2	9	--
14	ADI	gpr	xbd	left	yes	4	10	--
15	STR	gpr	---	---	no	4	11	--
16	GRT	gpr	gpr	cc	no	2	12	--
17	GRT	gpr	xbd	cc	yes	4	13	--
18	OR	gpr	gpr	left	no	2	14	--
19	OR	gpr	gpr	cc	no	2	14	--
20	OR	gpr	cc	left	no	8	15	--
21	OR	scb	scb	scb	no	0	0	--
22	FJP	scb	---	---	no	0	0	--

Table 3.3. Cost Table Entries Required by Algorithm.

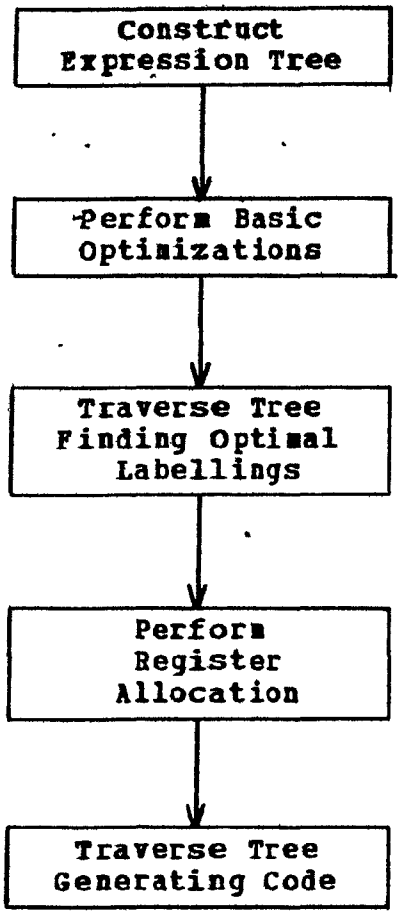


Figure 3.5. Code Generation Modules.

Chapter 4

Other Implementations

In this chapter we will describe two other projected implementations. One of them is for the PDP-11 and the other one deals with the VAX-11. We have chosen these machines for two reasons. First, the PDP-11 is very different from the IBM 360/370 and thus poses several problems that must be dealt with. The VAX-11 is also quite different as its architecture posesses features that are absent on both other machines. The second reason for choosing the PDP-11 and the VAX-11 is that they are both widely used and it therefore makes sense to demonstrate our code generation method for these machines.

4.1 PDP-11 Implementation.

As a second implementation we chose the PDP-11 [24]. Of all the addressing modes available for accessing main memory on the PDP-11, only three are used by the code generator.

Addressing modes such as autoincrement and autodecrement are not needed by the code generator as there are no P-code operations, that, taken individually, correspond to these addressing modes combined with another P-operation such as add, subtract, etc. In our code generator model, it is up to the peephole optimizer to introduce uses of the autoincrement or autodecrement modes between code templates whenever possible. These addressing modes are used within some code templates, however, (mainly for certain P-operations on sets or for some move operations). The storage classes selected to represent the three addressing modes are IR for "Indirect Register", RX for "Indexed Register" and IRX for "Indirect Indexed Register". A summary of the possible storage classes and conversion templates is given in Table 4.1.

The PDP-11 has two important differences from the IBM 360/370. The first is that most instructions have a two-address format and therefore destroy the value held in one of the operands. With the IBM 360/370, the operand destroyed is almost always held in a register and it was implicitly assumed in our compiler model that registers are used only as temporaries and could therefore be reused. With the PDP-11 the operand could be either register or memory. If it is memory, we insist (unless optimization analysis proves otherwise) that this memory be a temporary location generated by the compiler. Therefore, a new storage class, TMEM, which refers to a temporary memory location and which can be provided as an

input storage class to destructive instructions, is required. Since the use of a register is more efficient than the use of a memory location, the TMEM storage class will generally only be used when there are no available registers. Another use of the memory class as a destination operand could arise when the destination operand is one of the source operands. Consider for example the Pascal statement A := A + B. In this statement, it is more efficient to add the contents of B directly to the contents of A in memory rather than to use a register. Although, in this case, A does not represent a temporary memory location but rather a memory location that has been determined to be reusable, for the purpose of the code generator it can be treated as such. The information indicating that a memory location is reusable should be supplied to the code generation phase by a prior optimization analysis phase. Since the TMEM storage class is to be exclusively used in the two cases mentioned above (i.e. it will never be obtained from the code template selection algorithm or from cost table entries), there is no provision for converting from another storage class to the TMEM class. A sample expression tree using the TMEM storage class is given in Figure 4.1. Care must be taken, however, in the way the TMEM storage class is used. In the following statement A := A + (A + B), one must be careful not to assign the result of A + B to the TMEM class that corresponds to the storage location for A (as was done in the previous example) because the value of A is still needed in the expression.

Recommended Storage Classes for PDP-11

REG	General Register (R0-R5)
FPR	Floating Point Register
IR	Memory (Indirect Addressing Mode)
RX	Memory (Indexed Addressing Mode)
IRX	Memory (Indirect Indexed Mode)
IMM	Immediate Constant
OPND	Operand (REG/IR/RX/IRX/IMM class)
CC	Condition Code
SCB	Short-Circuit Boolean
TMEM	Temporary Memory Location
RXA	Address Constant (RX format)

Elementary Conversion Templates

REG	-->	OPND	{ No code generated }
IR	-->	OPND	{ No code generated }
RX	-->	OPND	{ No code generated }
IRX	-->	OPND	{ No code generated }
IMM	-->	OPND	{ No code generated }
OPND	-->	REG	{ MOV instruction }
TMEM	-->	REG	{ MOV instruction }
CC	-->	REG	{ MOV,BR,MOV sequence }
SCB	-->	REG	{ MOV,BR,MOV sequence }
CC	-->	SCB	{ BGE,... instruction }
OPND	-->	CC	{ TST instruction }

Table 4.1. PDP-11 Storage Classes and Conversion Templates.

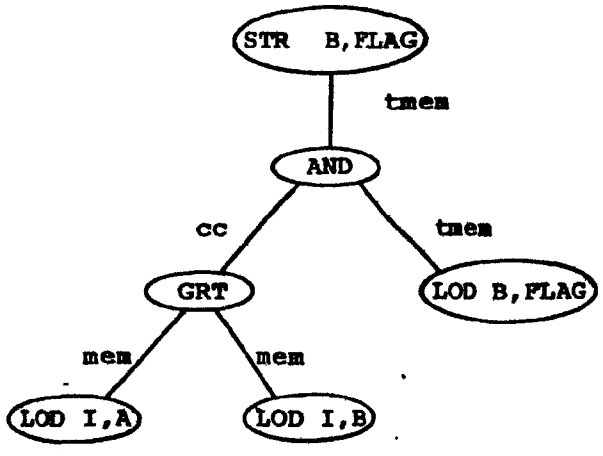


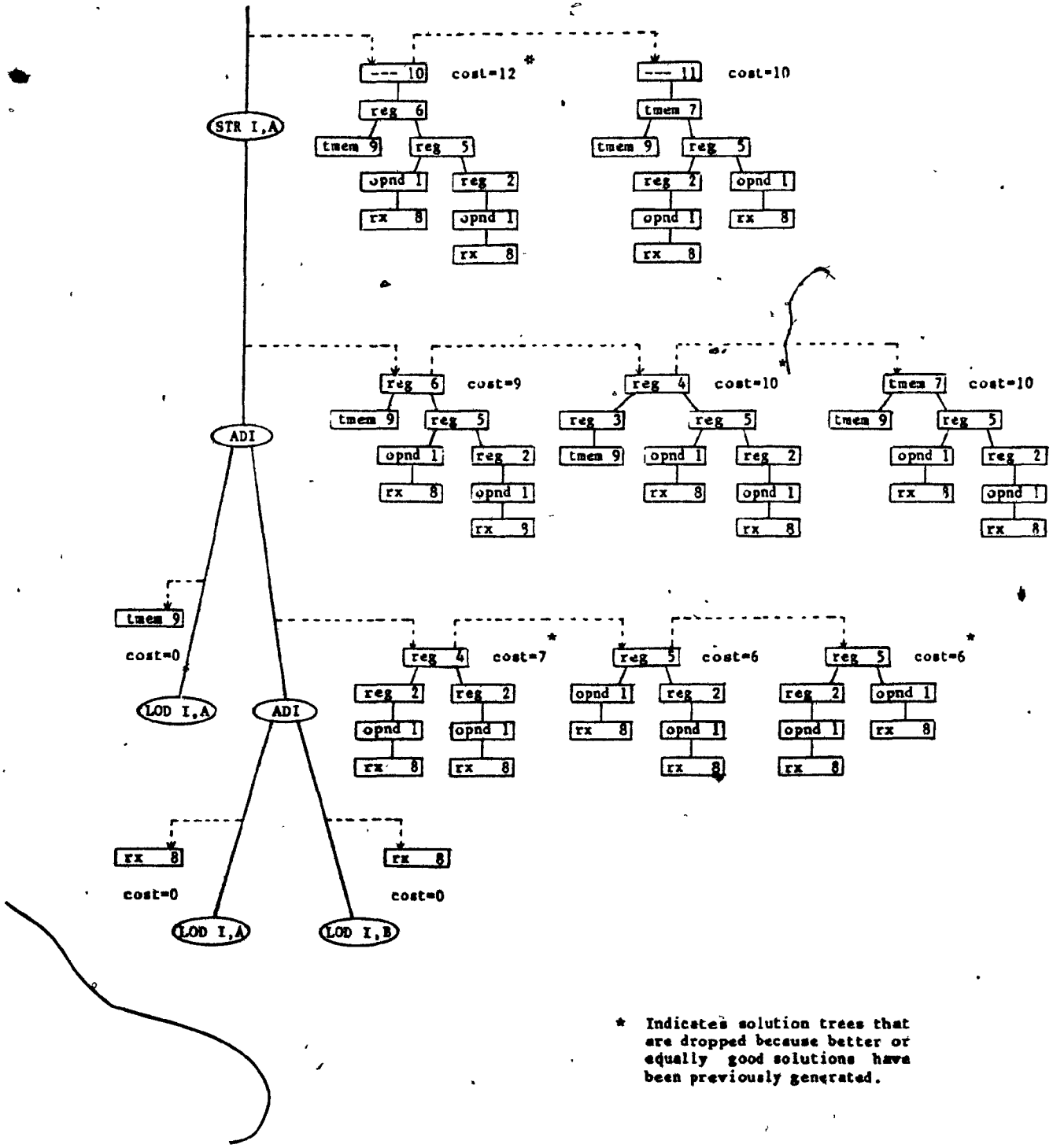
Figure 4.1. Expression Tree Labelled with PDP-11 Storage Classes for Pascal Statement FLAG := (A > B) and FLAG.

The second difference involving the PDP-11 is that most instructions accept operands that can have any addressing mode. Since different storage classes were allocated to different addressing modes, the implication is that distinct templates are needed for one instruction operating on different storage classes. Yet all the templates for the ADI P-code operation, say, would generate the same PDP-11 instruction, ADD. To eliminate the obvious redundancy, one more storage class OPND, which denotes an operand with an arbitrary storage class (out of those suitable for direct use in a PDP-11 instruction), is added. Conversions from REG, IR, RX, etc. to OPND (which do not actually generate code) are provided. With this technique, the number of templates for addition is greatly reduced. However, this also means that different costs cannot be attached to the different uses of the ADD instruction (i.e. the cost of using ADD on a memory operand as opposed to a register operand must be the same). Instead, the costs of using a particular kind of addressing mode must be attached to the appropriate template that provides the conversion to the OPND class. Here is a summary of all the templates that would be needed for integer addition, ADI:

```
REG  + TMEM  -->  REG   {Commutative}
OPND + REG   -->  REG   {Commutative}
OPND + TMEM  -->  TMEM  {Ccmmutative}
```

Since conversions between REG, TMEM and OPND are all available, this completes the set of templates. Figure 4.2 shows a sample tree along with its solutions as produced by

the code template selection algorithm. The cost table entries required for this tree are given in Table 4.2.



* Indicates solution trees that are dropped because better or equally good solutions have been previously generated.

Figure 4.2. Solution Tree for Pascal Statement A := A + (A + B) on the PDP-11.

ENT.#	OPER.	LEFT	RIGHT	RESULT	COMMUT.	COST	TEMP.#	TEST
1	CNV	rx	---	opnd	no	3	0	--
2	CNV	opnd	---	reg	no	2	1	--
3	CNV	tmem	---	reg	no	4	1	--
4	ADD	reg	reg	left	no	3	2	--
5	ADD	opnd	reg	right	yes	2	2	--
6	ADD	reg	tmem	left	yes	4	2	--
7	ADD	reg	tmem	right	yes	5	2	--
8	LOD	---	---	rx	no	0	0	--
9	LOD	---	---	tmem	no	0	0	reusable
10	STR	reg	---	---	no	3	1	--
11	STR	tmem	---	---	no	0	0	tmem= dest
12	STR	tmem	---	---	no	5	1	tmem<>dest

Table 4.2. Cost Table Entries Required by Code Template Selection Algorithm for the PDP-11.

4.2 VAX-11 Implementation.

The third implementation deals with the VAX-11 [25]. Among the addressing modes available for accessing main memory on the VAX-11, ten are used by our code generator. The storage classes selected to represent these addressing modes are RD for "Register Deferred", RDX for "Register Deferred Indexed", BD, BDX, BDD, and BDDX for "Byte Displacement", "Byte Displacement Indexed", "Byte Displacement Deferred" and "Byte Displacement Deferred Indexed", respectively. In addition the WD, WDX, WDD and WDDX storage classes represent the word displacement equivalents of the BD, BDX BDD and BDDX classes. A summary of the possible storage classes and conversions templates for this machine is given in Table 4.3.

Recommended Storage Classes for VAX-11

REG	-	General Register (R0-R13)
RD	-	Memory (Register Deferred)
RDX	-	Memory (Register Def. Ind.)
BD	-	Memory (Byte Displacement)
BDX	-	Memory (Byte Disp. Indexed)
BDD	-	Memory (Byte Disp. Deferred)
BDDX	-	Memory (Byte Disp. Def. Ind.)
WD	-	Memory (Word Displacement)
WDX	-	Memory (Word Disp. Indexed)
WDD	-	Memory (Word Disp. Deffered)
WDDX	-	Memory (Word Disp. Def. Ind.)
IMM	-	Immediate Constant
OPND	-	Operand (REG/RD/.../IMM class)
TREG	-	Temporary Internal Register
CC	-	Condition Code
SCB	-	Short-Circuit Boolean
TMEM	-	Temporary Memory Location
RDA	-	Address Constant (RD format)
BDA	-	Address Constant (BD format)
WDA	-	Address Constant (WD format)

Elementary Conversion Templates

REG	-->	OPND	{ No code generated }
RD	-->	OPND	{ No code generated }
RDX	-->	OPND	{ No code generated }
BD	-->	OPND	{ No code generated }
BDX	-->	OPND	{ No code generated }
BDD	-->	OPND	{ No code generated }
BDDX	-->	OPND	{ No code generated }
WD	-->	OPND	{ No code generated }
WDX	-->	OPND	{ No code generated }
WDD	-->	OPND	{ No code generated }
WDDX	-->	OPND	{ No code generated }
IMM	-->	OPND	{ No code generated }
TREG	-->	REG	{ No code generated }
TREG	-->	TMEM	{ No code generated }
OPND	-->	REG	{ MOV instruction }
TMEM	-->	REG	{ MOV instruction }
CC	-->	REG	{ MOV,BR,MOV sequence }
SCB	-->	REG	{ MOV,BR,MOV sequence }
CC	-->	SCB	{ BGE ... instruction }
OPND	-->	CC	{ TST instruction }

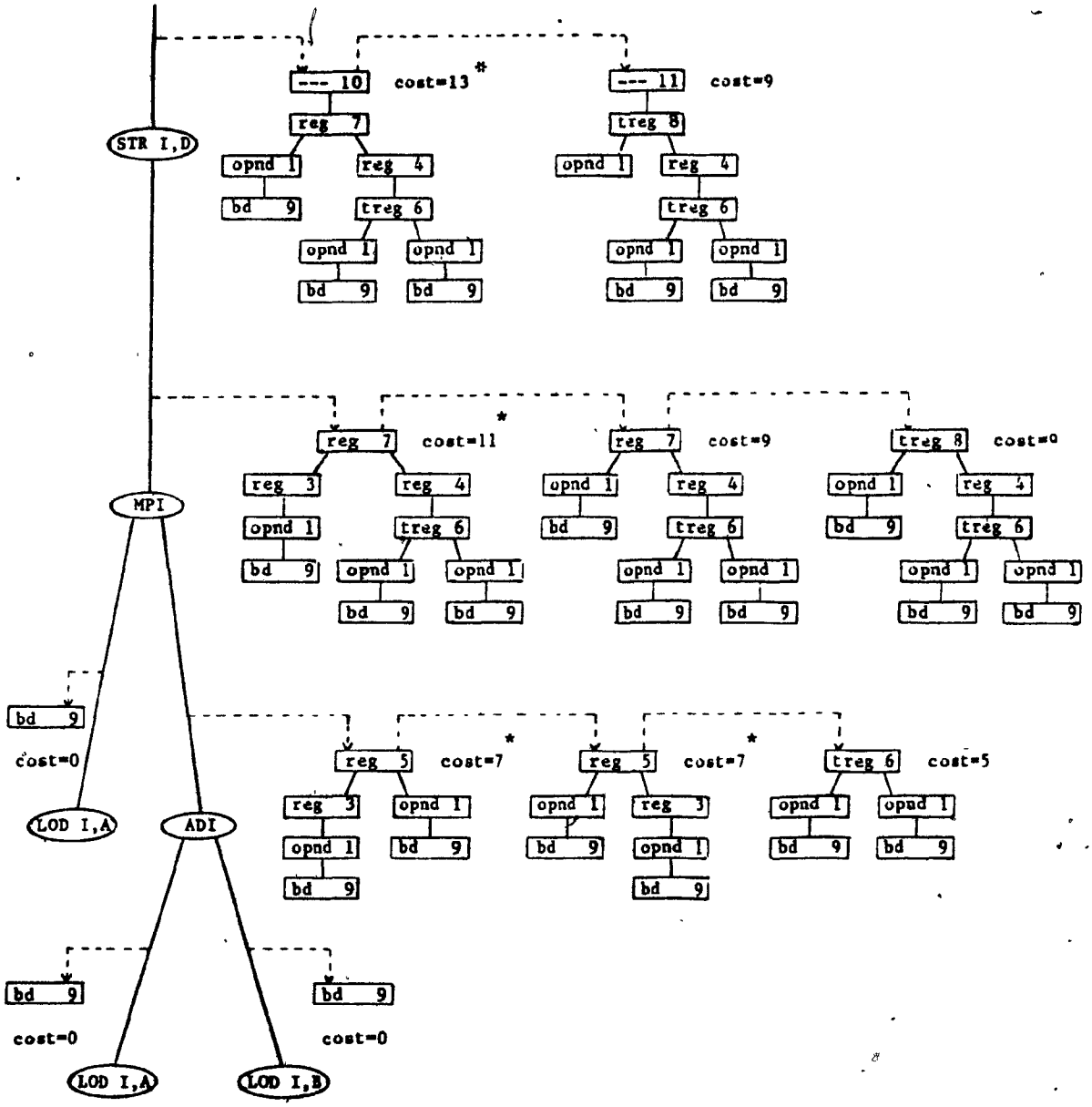
Table 4.3. VAX-11 Storage Classes and Conversion Templates.

The VAX-11 is similar to the PDP-11 in that it can modify operands held in memory. For this reason, a temporary memory class is also required for this machine. The TMEM storage class is used in the same way as with the PDP-11. Here again a general storage class, OPND, is used to denote an operand with an arbitrary storage class. One important difference from the PDP-11 and the IBM 360/370 is that certain instructions on the VAX-11 allow a destination operand that is different from either of its source operands. That is, it has some three address format instructions [26]. In the case of the PDP or the IBM, the result of an instruction is almost always placed at the same location as one of its source operands. The destination operand can be taken advantage of to combine the effects of a P-code operation followed by a conversion from the resulting storage class to the class required as an operand for a subsequent operation. The destination operand is handled by a new storage class called TREG for "temporary register". The TREG class can be viewed as an internal register used to hold the result of an instruction until the destination operand has been determined. Conversions are provided from the TREG class to all other storage classes. This facility can greatly reduce the number of conversions, between storage classes, that are normally required for both the IBM and the PDP. The destination operand provides a great mechanism for chaining instructions together to obtain a code sequence for an entire expression tree. It can also be used to combine

the effects of a P-code operation followed by a "store P-instruction". A sample expression tree is given in Figure 4.3. The cost table entries required for this tree are given in Table 4.4.

ENT.#	OPER.	LEFT	RIGHT	RESULT	COMMUT.	COST	TEMP.#	TEST
1	CNV	bd	---	opnd	no	2	0	--
2	CNV	reg	---	opnd	no	1	0	--
3	CNV	opnd	---	reg	no	2	1	--
4	CNV	treg	---	reg	no	1	0	--
5	ADI	reg	opnd	left	yes	1	2	--
6	ADI	opnd	opnd	treg	no	1	3	--
7	MPI	reg	opnd	left	yes	1	4	--
8	MPI	opnd	opnd	treg	no	1	5	--
9	LOD	---	---	bd	no	0	0	--
10	STR	reg	---	---	no	4	1	--
11	STR	treg	---	---	no	2	0	--

Table 4.4. Cost Table Entries Required by Code Template Selection Algorithm for the VAX-11.



* Indicates solution trees that are dropped because better or equally good solutions have been previously generated.

Figure 4.3. Solution Tree for Pascal Statement D := A * (B + C) on the VAX-11.

4.3 Other Implementations.

It is quite certain that other computer architectures will require different storage classes. But, as it has been demonstrated in this chapter and the previous one, these storage classes are generally easy to obtain from the machine architecture. For this reason the selection of storage classes and the construction of the cost table should not require a great amount of work. This will be especially true if tables already exist for a different machine as they will remain similar from one implementation to another. The code template selection algorithm does not require any changes for either of the two implementations discussed in this chapter. It is possible, however, that for certain machine architectures the algorithm might require some minor changes. As an example consider a machine with very few registers. Such machines are quite common when one considers micro-processors. Since there are few registers, it will not be uncommon to run out of registers during the register allocation phase of the compiler. For this reason, it may be good idea to treat each register as a separate storage class. The code template selection algorithm should then be changed to reject tree labellings with two concurrently active uses of the same register class.

Chapter 5

Practical Results

The theory and notation developed in the previous chapters was used to construct a code generator for the Pascal language. The target machine selected was the IBM 370. The IBM computer was picked for the trial implementation because of its general availability and the author's familiarity with it. Pascal was used for the source language because of its relatively clean design, increasing popularity and the existence of Pascal compilers locally. The code generator contains several major components: a tree construction component which builds expression trees from the P-code operations read in; an optimizer which performs simple optimizations such as constant folding; the code template selection algorithm component which finds the best code sequence for an entire expression tree; a register allocation component; and finally the code generation component. Each component performs only the task it is assigned and all

components are independent from each other. The only link between them is the expression tree which is successively passed from one component to the other. In order to produce code of a better quality, the register allocation algorithm used is somewhat more complex than the one discussed in Chapter 3. It remembers the contents of registers within basic blocks, thus avoiding subsequent unnecessary loads [27].

When the implementation was first started, one of our major concerns was the eventual size of the tables used by the code template selection algorithm. As it turns out, however, the amount of storage required for the tables is small in comparison to the storage used by the equivalent source level routines. The table sizes are given in Table 5.1. Two tables provide cross references between the major tables. The size of the template table could be reduced by a considerable amount if special data structures were used to take advantage of duplicated code sequences occuring in templates. But, since the total amount of storage required for the tables is just over 5K bytes there is no need to use complex data structures. Note that, unlike other experimental implementations which only deal with arithmetic operations, our implementation accepts the complete Pascal language including all the set operations. The table sizes would increase slightly if new templates were added to improve on the quality of the code generated. As it stands now, the quality of the code generated is comparable to that of a

production quality code generator in a non-optimizing compiler and just as good as the code produced by other automatic code generation methods (as will be demonstrated later).

Table Name	Entries	Size of Entry	Total Bytes
Cost Table	258	10 bytes	2580
Conversion Table	144	4 bytes	576
Template Table	180	10 bytes	1800
Cross Reference 1	80	2 bytes	160
Cross Reference 2	180	2 bytes	360
Total Size:			5476

Table 5.1. IBM 370 Code Generator Table Sizes.

Another concern of ours was the amount of processing that would be required to determine the best code sequence. We have found that the average number of templates for any single node is about 1.25. The maximum number of templates, which occurs in the case of the AND and OR operations, is 4. Measurements show that the innermost loop of the template selection algorithm is executed 1.32 times on average per recursive call to the algorithm. The number of recursive calls depends solely on the complexity of the expression, but the average number is quite small for typical programs. The net result is that the amount of processing required to determine the best code sequence is not as great as was anticipated and is a small price to pay for good quality code.

```
{ Pascal Program }

const MAXINDEX = 9;
type MATRIX = array [0..MAXINDEX] of
                array [0..MAXINDEX] of integer;

procedure MATRIXMULT (var  A,B,C : MATRIX);

var  I, J, K, SUM : integer;

begin
  for I := 0 to MAXINDEX do
    for J := 0 to MAXINDEX do begin
      SUM := 0;
      for K := 0 to MAXINDEX do
        SUM := SUM + A[I,K] * B[K,J];
      C[I,J] := SUM;
    end;
end;  { MATRIXMULT }
```

```
{ Pascal Program }

var CH : char;

function READN : integer;

var LVAL, BASE : integer;

begin
  while CH = ' ' do read(CH);
  if (CH <= '9') and (CH >= '0') then
    begin
      if CH = '0' then BASE := 8
      else BASE := 10;
      LVAL := 0;
      repeat
        LVAL := LVAL * BASE + ord(CH) - ord('0');
        read(CH);
      until (CH < '0') or (ord(CH) - ord('0') >= BASE);
      READN := LVAL;
    end
  else READN := -1;
end;  { READN }
```

Figure 5.1. Pascal Routines Used for Code Comparisons.

In the remainder of this chapter we will compare the code generated by our method with the code produced by Glanville's machine independent code generator [18]. For this purpose, the same two programs that were used by Glanville were also used here. They appear in Figure 5.1. For comparison purposes, the register allocation algorithm used by our code generator was replaced by a simpler, more straightforward, one. Only code for the body of the procedures is used in the comparisons as was done by Glanville. The assembly code produced by our code generator and by Glanville's machine independent code generator for the MATRIXMULT procedure is shown in Figures 5.2 and 5.3, respectively. Aside from some obvious errors[1] that are present in Glanville's code, the two code sequences are quite similar. The code produced by our code generator consists of 53 instructions occupying 198 bytes plus 3 halfword constants for a total of 204 bytes of storage. The equivalent code from Glanville's machine independent code generator consists of 60 instructions and occupies 212 bytes. The difference in size can be partly attributed to the poor usage of the even/odd register pair by Glanville's code generator that is made apparent by the use of an unnecessary

---

[1] The constants used by Glanville for the array component lengths assume halfword integers while the array elements are accessed using fullword instructions. Also, the result of a multiply instruction is held in a register pair of which only the odd register is required for subsequent operations (the even register normally only contains the sign extension of the result) as opposed to the even register as used by Glanville.

LR instruction before each multiply instruction. Our code generator avoided the problem by using the more efficient multiply halfword instruction. The even/odd register pair is efficiently used in our code generator as demonstrated further down in the code and in the code produced for the READN procedure. It should be noted, however, that our register allocation algorithm is partly responsible for this improvement.

The use of literal constants in our code is also an improvement over the use of an LA instruction followed by a register to register instruction, as produced by Glanville's code generator. Our code sequence requires a 4 byte instruction plus a 2 byte data constant, whereas Glanville's code requires 6 bytes of instructions. Thus, cur approach is more efficient in storage if the data constant is referenced more than once (as it usually is). The difference in instruction timings is also negligible. The code issued by our code generator for array indexing could be improved if the indexing method initated the one used by Glanville's code generator. In so doing, 2 bytes would be saved for each array reference thus reducing the size of the code by 6 bytes. However, the P-code operations used for array indexing in our compiler make it difficult (but not impossible) for us to produce the same code sequence as Glanville's.

The assembly language output for both BEADN routines appears in Figure 5.4. Our code generator produced 36 instructions (excluding the calls to the read routine) requiring a total of 132 bytes of storage. Glanville's code generator produced 45 instructions requiring 164 bytes of memory. The difference stems from the fact that the global character CH occupies a single byte in our compiler and can thus be compared with constants without first being loaded into a register (by making use of SI format instructions). The allocation of a fullword to a character value can provide an advantage in Glanville's code when this character is used in some kinds of computation. In this case the character can be efficiently loaded into a register using an L instruction while in our case the less efficient SR and IC code sequence must be used. Since the use of characters in computations is fairly restrained, it is usually more efficient to allocate a single byte per character. Note that in the code sequences produced for both these routines, additional code improvements are possible by simply using a good peephole optimizer.

Additional comparisons were made between the code produced by our code generator and the code generated by the IBM PASCAL/VS compiler. For these comparisons the full capabilities of our register allocation algorithm were used as it was not known how much optimization is performed by the PASCAL/VS compiler. Two programs were selected for these comparisons. The first one is a simple tree traversal program

consisting of 114 lines of source code. The IBM compiler generated 1436 bytes of code while our code generator produced 1320 bytes. The second program is a simple statistical analysis program of 430 lines. For this program the IBM compiler generated 9444 bytes of code while our code generator produced 7316 bytes. The results obtained are very satisfying as our code generator performed extremely well in the comparisons, especially since IBM claims that their compiler performs optimizations.

```
        SR    R8,R8
        ST    R8,I(R13)
L2:     SR    R8,R8
        ST    R8,J(R13)
L4:     SR    R8,R8
        ST    R8,SUM(R13)
        SR    R8,R8
        ST    R8,K(R13)
L6:     L     R8,A(R13)
        L     R7,I(R13)
        MH    R7,=H'40'
        AR    R7,R8
        L     R6,K(R13)
        SLA   R6,2
        L     R7,0(R6,R7)
        L     R5,B(R13)
        L     R4,K(R13)
        MH    R4,=H'40'
        AR    R4,R5
        L     R3,J(R13)
        SLA   R3,2
        M     R6,0(R3,R4)
        A     R7,SUM(R13)
        ST    R7,SUM(R13)
        L     R8,K(R13)
        CH    R8,=H'9'
        BEQ   L7
        L     R8,K(R13)
        AH    R8,=H'1'
        ST    R8,K(R13)
        B     L6
L7:     L     R8,C(R13)
        L     R7,I(R13)
        MH    R7,=H'40'
        AR    R7,R8
        L     R6,J(R13)
        SLA   R6,2
        L     R5,SUM(R13)
        ST    R5,0(R6,R7)
        L     R4,J(R13)
        CH    R4,=H'9'
        BEQ   L5
        L     R3,J(R13)
        AH    R3,=H'1'
        ST    R3,J(R13)
        B     L4
L5:     L     R8,I(R13)
        CH    R8,=H'9'
        BEQ   L3
        L     R8,I(R13)
        AH    R8,=H'1'
        ST    R8,I(R13)
        B     L2
L3:
```

Figure 5.2. IBM 370 Assembly Code, Produced by our Code Generator, for the MATRIXMULT Routine.

```
        SR    R2,R2
        ST    R2,I(R14)
        BC    14,L2(R15)
L1:     SR    R2,R2
        ST    R2,J(R14)
        BC    14,L4(R15)
L3:     SR    R2,R2
        ST    R2,SUM(R14)
        SR    R2,R2
        ST    R2,K(R14)
        BC    14,L6(R15)
L5:     LA    R2,2
        LA    R3,10
        LR    R4,R3
        M     R4,I(R14)
        A     R4,K(R14)
        MR    R4,R2
        A     R4,A(R14)
        L     R2,0(R4)
        LA    R3,2
        LA    R4,10
        M     R4,K(R14)
        A     R4,J(R14)
        MR    R4,R2
        A     R4,B(R14)
        L     R3,0(R4)
        LR    R4,R3
        MR    R4,R2
        A     R4,SUM(R14)
        ST    R4,SUM(R14)
        LA    R2,1
        A     R2,K(R14)
        ST    R2,K(R14)
L6:     L     R2,K(R14)
        LA    R3,9
        CR    R2,R3
        BC    12,L5(R15)
        LA    R2,2
        LA    R3,12
        LR    R4,R3
        M     R4,I(R14)
        A     R4,J(R14)
        MR    R4,R2
        A     R4,C(R14)
        L     R2,SUM(R14)
        ST    R2,0(R4)
        LA    R2,1
        A     R2,J(R14)
        ST    R2,J(R14)
L4:     L     R2,J(R14)
        LA    R3,9
        CR    R2,R3
        BC    12,L3(R15)
        LA    R2,1
        A     R2,I(R14)
        ST    R2,I(R14)
L2:     L     R2,I(R14)
        LA    R3,9
        CR    R2,R3
        BC    12,L1(R15)
```

Figure 5.3. IBM 370 Assembly Code, Produced by Glanville's Machine Independent Code Generator, for MATRIXMULT.

```
L2:  CLI   CH(R12),C' '
     BNE   L3
      ... READ(CH) ...
     B     L2
L3:  CLI   CH(R12),C'9'
     BH    L4
     CLI   CH(R12),C'0'
     BL    L4
     CLI   CH(R12),C'0'
     BNE   L5
     LA    R8,8
     ST    R8,BASE(R13)
     B     L6
L5:  LA    R8,10
     ST    R8,BASE(R13)
L6:  SR    R8,R8
     ST    R8,LVAL(R13)
L7:  L     R7,LVAL(R13)
     M     R6,BASE(R13)
     SR    R8,R8
     IC    R8,CH(R12)
     AR    R7,R8
     SH    R7,=AL2(C'0')
     ST    R7,LVAL(R13)
      ... READ(CH) ...
     CLI   CH(R12),C'0'
     BL    L9
     SR    R8,R8
     IC    R8,CH(R12)
     SH    R8,=AL2(C'0')
     C     R8,BASE(R13)
     BL    L7
L9:  L     R8,LVAL(R13)
     ST    R8,READN(R13)
     B     L8
L4:  SR    R8,R8
     BCTR  R8,0
     ST    R8,READN(R13)
L8:
```

a) Our Code

```
L1:  L     R2,CH(R13)
     LA    R3,C' '
     CR    R2,R3
     BC    6,L2(R15)
      ... READ(CH) ...
     BC    14,L1(R15)
L2:  L     R2,CH(R13)
     LA    R3,C'9'
     CR    R2,R3
     BC    2,L3(R15)
     L     R2,CH(R13)
     LA    R3,C'0'
     CR    R2,R3
     BC    4,L3(R15)
     L     R2,CH(R13)
     LA    R3,C'0'
     CR    R2,R3
     BC    6,L5(R15)
     LA    R2,8
     ST    R2,BASE(R14)
     BC    14,L6(R15)
L5:  LA    R2,10
     ST    R2,BASE(R14)
L6:  SR    R2,R2
     ST    R2,LVAL(R14)
L7:  L     R2,BASE(R14)
     M     R2,LVAL(R14)
     A     R2,CH(R13)
     LA    R3,C'0'
     SR    R2,R3
     ST    R2,LVAL(R14)
      ... READ(CH) ...
     L     R2,CH(R13)
     LA    R3,C'0'
     CR    R2,R3
     BC    4,L8(R15)
     L     R2,CH(R13)
     LA    R3,C'0'
     SR    R2,R3
     C     R2,BASE(R14)
     BC    4,L7(R15)
L8:  L     R2,LVAL(R14)
     ST    R2,READN(R14)
     BC    14,L4(R15)
L3:  LA    R2,1
     LCR   R2,R2
     ST    R2,READN(R14)
L4:
```

b) Glanville's Code

Figure 5.4. IBM 370 Assembly Code for READN Routines.

Other complete implementations for different machines were not done in order to keep the work manageable by a single person. However, cost tables were constructed for both the PDP-11 and the VAX-11 using the storage classes presented in Chapter 4. Using these tables and the code template selection algorithm, both sample programs were hand translated. The code that was produced for the PDP-11 is shown in Figures 5.5 and 5.6 along with the corresponding code produced by Glanville's machine independent code generator. Note that here again the register pair is not properly used in the multiply instruction. The code obtained for the READN procedure, using our cost tables, is very similar to the code produced by Glanville's machine independent code generator, except for the use of byte instructions whenever the character CH is used as an operand. In doing so, an additional instruction was required to convert from a byte to a word for a subsequent addition. The code obtained for MATRIXMULT, using the tables, requires 67 words of memory compared to the 65 words required by Glanville's code generator. The difference can be attributed to the more efficient code that is generated for array indexing in Glanville's code generator. The intermediate language operations used for indexing make it difficult for us to produce the more efficient code pattern. Each array reference would require an additional 2 words if it were not for the use of the shift instruction to implement the doubling operation. Using this instruction, only 3 additional words are required for all 3 array references. The remainder

of the code is very similar. Glanville also noted the possibility of using the shift instruction and subsequently provided an additional production to produce that code pattern.

The branch instruction on the PDP-11 has a limited range of 128 words forward or backward. Whenever the destination of a branch instruction is more than 128 words away, the JMP instruction must be used. In the code obtained for both sample routines, the use of JBR, JNE, JGT, etc..., represent macro calls which expand to BNE or BEQ *+2 followed by JMP, for the BNE case, depending on how far away the destination is. In our sample routines, the JMP instruction is not required as all branches are over short distances.

Similarly, both test programs were hand translated using the tables for the VAX-11. The code that was produced appears in Figure 5.7. The code for the MATRIXMULT procedure consists of 28 instructions requiring 112 bytes of storage. Better code could be produced, particularly for array indexing, if the intermediate P-code was modified so as to be better able to take advantage of the sophisticated indexing instructions available on the VAX-11. The code obtained for the READN routine is similar to the code produced for the PDP-11 except that three-address instructions were used whenever possible. The READN routine required 29 instructions for a total of 115 bytes of storage. Unfortunately, no VAX was readily available

to compare our code with the code produced by one of their compilers. However, we include the code for the VAX-11 to show the applicability of our code generator to a wide range of target machines and to show the ease of retargeting the code generator.

```
L2:  CMPB CH,#' '
     JNE  L3
     JSR  PC,GETCHAR
     JBR  L2
L3:  CMPB CH,#'9'
     JGT  L4
     CMPB CH,#'0'
     JLT  L4
     CMPB CH,#'0'
     JNE  L5
     MOV  #8,4(R5)
     JBR  L6
L5:  MOV  #10,4(R5)
L6:  CLR  2(R5)
L7:  MOV  2(R5),R1
     MUL  4(R5),R1
     MOVB CH,R0
     ADD  R0,R1
     SUB  #'0',R1
     MOV  R1,2(R5)
     JSR  PC,GETCHAR
     MOVB R0,CH
     CMPB CH,#'0'
     JLT  L9
     MOVB CH,R1
     SUB  #'0',R1
     CMP  R1,4(R5)
     JLT  L7
L9:  MOV  2(R5),0(R5)
     JBR  L8
L4:  MOV  #-1,0(R5)
L8:  MOV  0(R5),R0
```

a) Our Code

```
L1:  CMP  CH,$40
     LNE  L2
     JSR  PC,GETCHAR
     JBR  L1
L2:  CMP  CH,$71
     JGT  L3
     CMP  CH,$60
     JLT  L3
     CMP  CH,$60
     JNE  L5
     MOV  $10,-6(R5)
     JBR  L6
L5:  MOV  $12,-6(R5)
L6:  CLR  -4(R5)
L7:  MOV  -6(R5),R0
     MUL  -4(R5),R0
     ADD  CH,R0
     SUB  $60,-4(R5)
     MOV  R0,-4(R5)
     JSR  PC,GETCHAR
     MOV  R0,CH
     CMP  CH,$60
     JLT  L8
     MOV  CH,R0
     SUB  $60,R0
     CMP  R0,-6(R5)
     JLT  L7
L8:  MOV  -4(R5),-2(R5)
     JBR  L4
L3:  MOV  $-1,-2(R5)
L4:  MOV  -2(R5),R0
```

b) Glanville's Code

Figure 5.5. PDP-11 Assembly Code for READN Routines.

```
        CLR   6(R5)
L2:     CLR   8(R5)
L4:     CLR   12(R5)
        CLR   10(R5)
L6:     MOV   (R5),R0
        MOV   6(R5),R1
        MUL   #10,R1
        ADD   (R0),R1
        MOV   2(R5),R2
        ASL   R0
        ADD   R1,R0
        MOV   2(R5),R2
        MOV   10(R5),R1
        MUL   #10,R1
        ADD   (R2),R1
        MOV   8(R5),R2
        ASL   R2
        ADD   R1,R2
        MOV   (R2),R1
        MUL   (R0),R1
        ADD   R1,12(R5)
        CMP   10(R5),#9
        JGT   L7
        INC   10(R5)
        JBR   L6
L7:     MOV   4(R5),R0
        MOV   6(R5),R1
        MUL   #20,R1
        ADD   (R0),R1
        MOV   8(R5),R0
        ASL   R0
        ADD   R1,R0
        MOV   12(R5),(R0)
        CMP   8(R5),#9
        JGT   L5
        INC   8(R5)
        JBR   L4
L5:     CMP   6(R5),#9
        JGT   L3
        INC   6(R5)
        JBR   L2
```

a) Our Code

```
        CLR   -2(R5)
        JBR   L2
L1:     CLR   -4(R5)
        JBR   I4
L3:     CLR   -10(R5)
        CLR   -6(R5)
        JBR   I6
L5:     MOV   $12,R0
        MUL   -2(R5),R0
        ADD   -6(R5),R0
        MUL   $2,R0
        ADD   -12(R5),R0
        MOV   $12,R1
        MUL   -6(R5),R1
        ADD   -4(R5),R1
        MUL   $2,R1
        ADD   10(R5),R1
        MOV   (R1),R1
        MUL   (R0),R1
        ADD   R1,-10(R5)
        INC   -6(R5)
L6:     CMP   -6(R5),$11
        JLE   L5
        MOV   $12,R0
        MUL   -2(R5),R0
        ADD   -4(R5),R0
        MUL   $2,R0
        ADD   -6(R5),R0
        MOV   -10(R5),(R0)
        INC   -4(R5)
L4:     CMP   -4(R5),$11
        JLE   L3
        INC   -2(R5)
L2:     CMP   -2(R5),$11
        JLE   L1
```

b) Glanville's Code

Figure 5.6. PDP-11 Assembly Code for MATRIXMULT Routines.

```
    CLRW  I(R13)
L2: CLRW  J(R13)
L4: CLRW  SUM(R13)
    CLRW  K(R13)
L6: MULW3 #20,I(R13),R1
    ADDW2 A(R13),R1
    MOVW  K(R13),R2
    MULW3 #20,K(R13),R3
    ADDW2 B(R13),R3
    MOVW  J(R13),R4
    MULW3 (R1)[R2],(R3)[R4],R1
    ADDW2 R1,SUM(R13)
    CMPW  K(R13),#9
    BGTR  L7
    INCW  K(R13)
    BRW   L6
L7: MULW3 #20,I(R13),R1
    ADDW2 C(R13),R1
    MOVW  J(R13),R2
    MOVW  SUM(R13),(R1)[R2]
    CMPW  J(R13),#9
    BGTR  L5
    INCW  J(R13)
    BRW   L4
L5: CMPW  I(R13),#9
    BGTR  L3
    INCW  I(R13)
    BRW   L2
```

a) MATRIXMULT Routine

```
L2: CMPB  CH(R12),#' '
    BNEQ  L3
    BSBW  PC,GETCHAR
    MOVB  R0,CH(R12)
    BRW   L2
L3: CMPB  CH(R12),#'9'
    BGTR  L4
    CMPB  CH(R12),#'0'
    BLSS  L4
    CMPB  CH(R12),#'0'
    BNEQ  L5
    MOVW  #8,BASE(R13)
    BRW   L6
    MOVW  #10,BASE(R13)
L6: CLRW  LVAL(R13)
L7: MULW3 LVAL(R13),BASE(R13),R1
    ADDB  CH(R12),R1
    SUBW3 #'0',R1,LVAL(R13)
    BSBW  PC,GETCHAR
    MOVB  R0,CH(R12)
    CMPB  CH(R12),#'0'
    BLSS  L9
    SUBB3 #'0',CH(R12),R1
    CMPW  R1,LVAL(R13)
L9: MOVW  BASE(R13),READN(R13)
    BRW   L8
L4: MOVW  #-1,READN(R13)
    MOVW  READN(R13),R0
L8:
```

b) READN Routine

Figure 5.7. VAX-11 Assembly Code.

The code comparisons should have demonstrated not only the effectiveness of our method of code generation, but also the high quality code that is generated and the ease with which this quality can be improved by adding additional code templates or new storage classes. In fact, in most cases, the addition of a new template simply consists of adding an extra entry to the cost table and inserting the actual template into

the template table. The ease with which new templates were added to our code generator during the trial implementation even surprised the author.

Chapter 6

Conclusion

The code generation method presented in this thesis represents a step towards automatic code generation. First the concept of a storage class was presented. Then, using this concept, an algorithm for selecting and joining code templates so as to produce near-optimal code was developed. The basic approach was shown to be quite machine independent. The quality of the code produced by our method is at least as good as, if not better than, what has been achieved by other automatic code generation methods. The tables used are easily constructed and their size is small in comparison to the equivalent source level routines. Furthermore, such a description represents a major improvement in clarity and modifiability. The ease with which new tables can be obtained and substituted for the existing ones, thus creating a cross compiler, is not present for standard code generation schemes and represents a major step towards facilitating the

retargeting of compilers. Similarly, it is straightforward to add new code templates to the tables either to implement new intermediate language operators or to take advantage of new instructions available on upward compatible computers in a computer series. The modularity of the code generator is also an improvement over currently existing code generators. It allows easy extensibility and modifiability, something that is not true of most code generators. This modularity allows us to treat register allocation as a totally independent phase and, therefore, it simplifies our task of designing and implementing our register allocation algorithm. The use of our register allocation algorithm provides a 15% reduction in the size of the generated code over a simple register allocation algorithm.

As with most code generation methods, this approach does not work too well with some of the more awkward instruction sets. Some computer architectures have too many exceptions such as special functions associated with specific registers or instructions that have different ways of handling addressing modes. However, as was demonstrated, our method is very flexible and can handle many intricacies that are present in most architectures. As an example, our method had no trouble in handling the use of the even/odd register pair on the IBM 360/370. In fact, our approach does seem to provide a greater flexibility in dealing with these machine intricacies

than some of the other automatic code generation methods. This is due to the ability of creating storage classes as needed to represent machine storage locations.

Several extensions to this research are possible. First the implementations for the PDP-11 and the VAX-11 should be completed. These implementations would provide further insight in our method of code generation and possibly reveal certain deficiencies or possible improvements that remain as yet unnoticed. Several enhancements have already been considered. First, a sophisticated optimization phase is currently under development and will eventually precede our code generation phase. Most of the standard optimizations are planned and others that deal with the more exotic features of the Pascal language, such as sets, are also considered [29]. The optimization phase might require changes in the intermediate language to allow further machine independence and to facilitate certain optimizations [30]. The intermediate language should definitely be modified to allow the code generator to take advantage of some of the more sophisticated instructions now available on the newer computers. A prime example would be the use of the INDEX and CASE instructions on the VAX-11. Although already quite sophisticated, several improvements to our register allocation algorithm are still possible. All these extensions will serve to improve the overall quality of the code generated and as

more and more system software is written in high level languages, this can become a major criterion in the selection of a language as a systems programming language.

None of the extensions mentioned so far will increase machine independence. However, considerable research is also possible in that area. One of the more promising possibilities is the automatic generation of the tables required by the code template selection algorithm from some kind of machine description. The register allocation algorithm could be made substantially more machine independent. In our current code generator the register allocation routines were imposed over our code template selection algorithm and thus are not very machine independent. These enhancements would greatly reduce the burden of retargeting the code generator. As new micro-processors become increasingly available, the ease with which a compiler can be retargeted becomes a major factor in determining whether it will be used or not.

Although much research remains to be done on automatic code generation, this thesis represents a step in the right direction. The approach taken in this thesis is significantly different from other methods taken in the past. Our method provides some distinct advantages. The two most important ones are the applicability of our approach to a wide range of computer architectures and the high quality code that is

produced. This research provides additional groundwork for future research in this area which may some day lead to totally automatic code generation. Any contribution in this area should be useful as it is not yet clear which is the best approach to automatic code generation.

BIBLIOGRAPHY

[1] Aho, A.V., and Ullman, J.D., Principles of Compiler Design, Addison Wesley, Reading, Mass., 1977.

[2] Barrett, William A., and Couch, John D., Compiler Construction: Theory and Practice, Computer Science Series, 1979.

[3] Gries, D., Compiler Construction for Digital Computers, Wiley, New York, 1971.

[4] McKeeman, W.M., Peephole Optimization, Comm. ACM 8,7, (July 1965), 443-444.

[5] Davidson, J.W., Fraser, Christopher W., The Design and Application of a Retargetable Peephole Optimizer, ACM Trans. on Programming Languages and Systems, 2,2 (April 1980), 191-202. Also see Corrigendum, 3,1 (January 1981), 110.

[6] Aho, A.V. and Johnson, S.C., Optimal Code Generation for Expression Trees, Journal of ACM 23,3, (July 1976), 488-501.

[7] Aho, A.V., Johnson, S.C. and Ullman, J.D., Code Generation for Expressions with Common Subexpressions, J. ACM 21,4, (January 1977), 146-160.

[8] Aho, A.V., Johnson, S.C. and Ullman, J.D., Code Generation for Machines with Multiregister Operations, Proc. Fourth ACM Symposium on Principles of Programming Languages, (1977), 21-28.

[9] Elson, M. and Rake, S.T., Code Generation Technique for Large Language compilers, IBM Syst. Journal 9,3, (1970), 166-188.

[10] Wilcox, T.R., Generating Machine Code for High-Level Programming Languages, Ph.D. Thesis, Cornell University, (Sept. 1971).

[11] Donegan, M.K., An Approach to the Automatic Generation of Code Generators, Ph.D. Thesis, Rice University, Houston, Texas, (May 1973).

[12] Miller, M.K., Automatic Creation of a Code Generator from a Machine Description, Technical Report MAC TR-85, M.I.T. Cambridge MA, (May 1971).

[13] Weingart, S.W., An Efficient and Systematic Method of Compiler Code Generation, Ph.D. Thesis, Computer Science Department, Yale University, (1973).

[14] Newcomer, J.M., Machine-Independent Generation of Optimal Local Code, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, (May 1975).

[15] Wulf, W.A., et. al., BLISS: A Basic Language for Implementation of System Software for the PDP-10, Carnegie-Mellon University CSD report, (1970).

[16] Wulf, W.A., et. al., BLISS-11: Programmer's Manual, Digital Equipment Corp., (1972).

[17] Fraser, C.W., Automatic Generation of Code Generators, Ph.D. Thesis, Yale University, (December 1977).

[18] Glanville, R.S., A Machine Independent Algorithm for Code Generation and its use in Retargetable Compilers, Ph.D. Thesis, Computer Sciences Division, University of California, Berkeley, (November 1977).

[19] Cattell, R.G.G., Automatic Derivation of Code Generation from Machine Descriptions, ACM Trans. on Programming Languages and Systems 2,2, (April 1980).

[20] Wulf, W.A., Compilers and Computer Architecture. Computer, 14,7, (July 1981), 41-48.

[21] Nori, K., Amman, U., Jensen, K., and Nagel, H., The Pascal P Compiler - Implementation Notes. Berichte des Instituts fur Informatik, E.T.H., Zurich (Dec. 1974).

[22] IBM System/360 Principles of Operation, IBM Corporation Manual (A22-6821-3), Poughkeepsie, N.Y., (1966).

[23] Harrison, William, A Class of Register Allocation Algorithms, IBM Researc Report RC 5342, (March 1975).

[24] PDP-11/45 Processor Handbook, Digital Equipment Corporation, (1972).

[25] VAX-11/780 Architecture Handbook, Digital Equipment Corporation, (1977-78).

[26] Stone, Harold S., et. al., Introduction to Computer Architecture, Science Research Associates, Inc., (1975).

[27] Beatty, J.C., Register Assignment Algorithm for Generation of Highly Optimized Object Code, IBM Journal of Research and Development, 18(1), (January 1974), 20-39.

[28] Agresti, William W., Register Assignment in Tree-Structured Programs, Information Sciences Journal, 18, (1979), 83-94.

[29] Horspool, R. Nigel, Dunkelman, Laurence N., Analysis and Optimization of Set Expressions, The Computer Journal, to appear, (1982).

[30] Ganapathi, Mahadevan, Fischer, Charles N., Scalpone, Stephen J., Thompson, Keith C., Linear Intermediate Representation for Portable Code Generation, Computer Sciences Technical Report #435, University of Wisconsin Madison, (September 1981).

Appendix A: Meanings of P-operations Used in Examples.

ADI : Integer Addition.
ADR : Real Addition.
SBI : Integer Subtraction.
SBR : Real Subtraction.
MPI : Integer Multiplication.
MPR : Real Multiplication.
DVI : Integer Division.
DVR : Real Division.
MOD : Remainder of Integer Division.
NGI : Integer Negation.
NGR : Real Negation.
ABI : Integer Absolute Value.
ABR : Real Absolute Value.
NOT : Logical Negation.
AND : Logical And.
OR : Logical Or.
IXA : Array Indexing. [1]
IND : Load Indirect. [2] [3]
LDA : Load Address. [4]
LOD : Load Data Item. [2] [4]
LDC : Load Constant. [2]
LCA : Load Constant Address.
STR : Store Data Item. [2] [4]
STO : Store Indirect. [2]
EQU : Comparison for Equal. [2]
NEQ : Comparison for Not Equal. [2]
LES : Comparison for Less. [2]
LEQ : Comparison for Less or Equal. [2]
GRT : Comparison for Greater. [2]
GEQ : Comparison for Greater or Equal. [2]
FJP : Jump if False.
UJP : Unconditional Jump.

Where
[1] : Array element length as an immediate operand.
[2] : Data type as an immediate operand.
[3] : Offset as an immediate operand.
[4] : Lexic level and offset as immediate operands.

Appendix B: IBM 360/370 Code Templates for P-operations of Appendix A.

```
ADI:  AR    R,R
      A     R,D(X,B)
      AH    R,D(X,B)

MPI:  MR    R,R
      M     R,D(X,B)
      MH    R,D(X,B)
      SLA   R,D

NGI:  LCR   R,R

ADR:  ADR   R,R
      AD    R,D(X,B)

MPR:  MDR   R,R
      MD    R,D(X,B)

NGR:  LCDR  R,R

AND:  NR    R,R

NOT:  X     R,=F'1'

FJP:  BC    mask,dest

IXA:  AR    R,R
      SLA   R,D
      MH    R,=F'k'

CHV:  SR    R,R
      SR    R,R;  BCTR R,R
      SR    R,R;  IC   R,D(X,B)
      LH    R,D(X,B)
      L     R,D(X,B)
      LD    R,D(X,B)
      LA    R,D(X,B)
      LR    R,R
      LTR   R,R
      SRDA  R,32
      AR    R,R
      LA    R,1;  BC   mask,*+6;  SR  R,R
      TM    D(B),1
      BC    mask,dest

SBI:  SR    R,R
      S     R,D(X,B)
      SH    R,D(X,B)

DVI,  DR    R,R
MOD:  D     R,D(X,B)

ABI:  LPR   R,R

SBR:  SDR   R,R
      SD    R,D(X,B)

DVR:  DDR   R,R
      DD    R,D(X,B)

ABR:  LPDR  R,R

OR:   OR    R,R

UJP:  BC    15,dest

STO:  STC   R,D(X,B)
      STH   R,D(X,B)
      ST    R,D(X,B)
      STD   R,D(X,B)

EQU:  CR    R,R
      LTR   R,R
      CH    R,D(X,B)
      C     R,D(X,B)
      CLI   D(B),I
      CDR   R,R
      CD    R,D(X,B)
      LTDR  R,R
      CLC   D(L,B),D(B)
```

Appendix C: PDP-11 Code Templates for P-operations of Appendix A.

```
ADI:  ADD  SRC,DST                 SBI:  SUB  SRC,DST
      INC  DST                           INC  DST
      DEC  DST                           DEC  DST

MPI:  MUL  SRC,REG                 DVI,
      ASH  DST,nn                  MOD:  DIV  SRC,REG

NGI:  NEG  DST

ABI:  TST  DST; BPL *+n; NEG DST

ADR:  ADDD SRC                     SBR:  SUBD SRC

MPR:  MULD SRC                     DVR:  DIVD SRC

NGR:  NEGD SRC                     ABR:  ABSD SRC

AND:  MOVB SRC,REG; COM REG;
      BITB SRC,REG

OR:   BIS  SRC,DST                 NOT:  COM  DST
      BISB SRC,DST                       COMB DST

FJP,  Bxx  dest
UJP:  JMP  dest

IXA:  ADD  REG,REG                 STO:  MOV  SRC,DST
      ASH  REG,nn                        MOVB SRC,DST
      MUL  SRC,REG                       STD  DST

CMV:  CLR  DST                     EQU:  CMP  SRC,SRC
      CLRB DST                           CMPB SRC,SRC
      CLRD DST                           TST  SRC
      MOV  SRC,DST                       TSTB SRC
      MOVB SRC,DST                       CMPD SRC
      LDD  SRC
      TSTB SRC
      MOVB #1,REG; Bcond *+2; CLRB REG
```

Where

SRC can be any addressing mode from amongst R, (R), X(R), X(R), and #n.

DST can be any addressing mode from amongst R, (R), X(R), X(R).

Appendix D: VAX-11 Code Templates for P-operations of Appendix A.

```
ADI:  ADDx2 SRC,DST                 SBI:  SUBx2 SRC,DST
      ADDx3 SRC,SRC,DST                   SUBx3 SRC,SRC,DST
      INCx  DST                           INCx  DST
      DECx  DST                           DECx  DST

MPI:  MULx2 SRC,DST                 DVI,  DIVx2 SRC,DST
      MULx3 SRC,SRC,DST             MOD:  DIVx3 SRC,SRC,DST
      ASHL  nn,SRC,DST

ABI:  TSTx  DST; BGEQ *+n;          NGI:  MNEGx SRC,DST
      MNEGx SRC,DST

ADR:  ADDD2 SRC,DST                 SBR:  SUBD2 SRC,DST
      ADDD3 SRC,SRC,DST                   SUBD3 SRC,SRC,DST

MPR:  MULD2 SRC,DST                 DVR:  DIVD2 SRC,DST
      MULD3 SRC,SRC,DST                   DIVD3 SRC,SRC,DST

ABR:  TSTD  DST; BGEQ *+n;          NGR:  MNEGD SRC
      MNEGD SRC,DST

AND:  MCOMB SRC,REG;                OR:   BISx2 SRC,DST
      BITB2 SRC,DST                       BISx3 SRC,SRC,DST

NOT:  MCOMB SRC,DST                 FJP,  Bxx   dest
                                    UJP:  JMP   dest

IXA:  ADDx3 SRC,SRC,DST             STO:  MOVx  SRC,DST
      MULx3 SRC,SRC,DST                   MOVD  SRC,DST
      ASHL  nn,SRC,DST

CNV:  CLRx  DST                     EQU:  CMPx  SRC,SRC
      MOVx  SRC,DST                       CMPDx SRC,SRC
      MOVD  SRC,DST                       TSTx  SRC
      TSTx  SRC                           TSTD  SRC
      BBS   nn,SRC,dest
      BBC   nn,SRC,dest
      MOVB  #1,REG; Bcond *+2; CLRB REG
```

Where

x can be B, W, L, F, D depending on the data type.

SRC can be any addressing mode from amongst
R, (R), (X)[R], BD(R), BD(R)[R],BD(R),BD(R)[R],
WD(R), WD(R)[R],WD(R),WD(R)[R], #n.

DST can be any addressing mode from amongst
R, (R), (X)[R], BD(R), BD(R)[R],BD(R),BD(R)[R],
WD(R), WD(R)[R],WD(R),WD(R)[R], #n.