Programming Techniques

S. L. Graham, R. L. Rivest Editors

Assembling Code for Machines with Span-Dependent Instructions

Thomas G. Szymanski
Princeton University

Many modern computers contain instructions whose lengths depend on the distance from a given instance of such an instruction to the operand of that instruction. This paper considers the problem of minimizing the lengths of programs for such machines. An efficient solution is presented for the case in which the operand of every such "span-dependent" instruction is either a label or an assembly-time expression of a certain restricted form. If this restriction is relaxed by allowing these operands to be more general assembly-time expressions, then the problem is shown to be NP-complete.

Key Words and Phrases: span-dependent instructions, variable-length addressing, code generation, assemblers, compilers, NP-complete, computational complexity.

CR Categories: 4.11, 4.12, 5.25

Research supported in part by NSF Grant MCS 74-21939 A01.

Author's address: Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J. 08540.

1. Introduction

Many modern computers contain instructions whose lengths are span-dependent in the sense that the amount of storage occupied by a given instance of such an instruction is determined by the distance from that instruction to its operand. Typically, a short form of an instruction can be used if the instruction's operand is "close" to the instruction, otherwise a long form of the instruction must be used. It is usually preferable to use the shorter form whenever possible in an effort to reduce both program length and execution time.

The problem of deciding when it is possible to use the shorter form of an instruction on such a machine is made nontrivial by the fact that the distance from an instruction to its operand depends on the lengths of the intervening instructions. These lengths can, of course, depend either directly or indirectly on the length selected for the original instruction. In order to simplify code generation for compilers and other assembly language programmers of such machines, it is desirable to relegate the choice between long and short forms of instructions to the assembler. It is the purpose of this paper to investigate the conditions under which this choice can be made optimally and efficiently. We shall only consider translation methods which preserve statement order, thus no rearrangement of code will be permitted. Previous work considering similar problems may be found in [3] and [6].

In an attempt to avoid cumbersome notation we shall present our results in the context of a specific computer: the Digital/Equipment Corporation PDP-11 [5]. However, it should be noted that our results apply to any computer containing *span-dependent instructions* (henceforth *sdi*'s) as specified formally in the following definition.

Definition: An instruction is said to be *span-dependent* if 1) the instruction exists in two forms of differing length, 2) the shorter form of such an instruction can be used at machine location m only if that instruction's operand has an address between $m + a$ and $m + b$ where a and b are fixed (and possibly negative) integer constants, 3) the longer form of such an instruction can always be used in place of a shorter form. □

Two additional examples of machines possessing *sdi*'s are the Motorola 6800 microprocessor and the IBM 1130. Undoubtedly, many other examples exist.

The PDP-11 contains two types of unconditional transfer instruction. The *branch* instruction (mnemonic **br**) is two bytes long and can only be used if the branch target is within approximately 254 bytes[1] of the branch instruction. The *jump* instruction (mnemonic **jmp**) is four bytes long but is unrestricted with respect to the location of its target.

Conditional transfer instructions on the PDP-11 are

[1] More precisely, a branch instruction at address m can have as its target any instruction whose address is between $m - 254$ and $m + 256$ inclusive.

two bytes long and subject to the same target restrictions as the branch instruction. Two such instructions are *branch-if-equal* and *branch-if-not-equal* (mnemonics **beq** and **bne** respectively). No conditional analog of the jump instruction exists on this machine and, accordingly, a conditional transfer to a distant target must be synthesized by using a conditional branch in conjunction with a jump instruction. For example, if X were close enough, then it would be permissible to use the instruction "**beq** Z". On the other hand, if X were remote, then we would have to use a sequence of instructions such as "**bne** skip; **jmp** X; skip: . . . ".

The PDP-11 assembler available under the UNIX operating system [7] supports *extended branch mnemonics* such as **jbr, jeq,** and **jne.** If the circumstances permit, these are translated into branch type instructions, that is, **br, beq,** and **bne.** Otherwise, they are treated as **jmp**'s or as conditional branches over a **jmp** as described above. It is easy to verify that these extended branch mnemonics may be regarded as *sdi*'s according to the definition given above.

The actual algorithm used in the UNIX assembler for translating programs containing *sdi*'s is a three-pass algorithm which, although not always succeeding in minimizing the length of an object program,[2] has very good average performance. In Section 2 of this paper we consider this and other natural, but suboptimal, algorithms. In Section 3 a two-pass algorithm will be presented which is guaranteed to produce the minimum length translation of a program containing span-dependent instructions. This algorithm requires that the operands of all *sdi*'s be restricted to assembly-time expressions of a certain type. If this restriction is relaxed, the problem of program size minimization becomes NP-complete, as will be proved in Section 4.

Throughout this paper the label of an assembly language statement will be separated from the statement proper by a colon. As usual, this label represents the run-time address of the labelled instruction. A period will denote the address of the statement containing it. Greek letters will be used to denote sequences of instructions. The *size* of a code sequence α, denoted $|\alpha|$, is the amount of storage occupied by α under some specified translation. The notation "$\Delta = n$" will be used to represent an arbitrary sequence of instructions which does not contain any *sdi*'s and which occupies exactly n bytes of storage.

It is traditional in the design of assembly languages to allow the programmer to specify the operands of a statement by means of an *assembly-time expression.* We assume that these expressions are built up from statement labels (including "."), integer constants, operators and parentheses in the usual manner. We also assume that the usual procedence and associativity rules are used to interpret expressions.

Expressions can be designated *absolute* or *relocatable* according to the following rules:

1. An integer constant is an absolute expression.
2. A label is a relocatable expression.
3. If r denotes a relocatable expression and a an absolute expression, then an expression of the form $a + r$ or $r \pm a$ is a relocatable expression; an expression of the form $a \pm a$ or $r - r$ is an absolute expression.

We assume that the operands of all *sdi*'s in a program are specified by relocatable expressions.

Any translation of a program containing n *sdi*'s can be uniquely specified by listing the set of *sdi*'s which have been translated in the long form. We call such a set a *selection set.* Given a program P and a selection set S, we can assign addresses to the instructions of P in the obvious way. If S is the empty set (alternatively, the set of all *sdi*'s in P) then the resulting address assignment is called the *minimum* (*maximum*) *address assignment.* It corresponds to choosing a short (long) translation for every *sdi* in the program.

Definition: Let S be a selection set for program P. The *span* of an instruction whose operand is the expression E is the value of the expression "$E-$." when evaluated under the address assignment determined by S. The translation of P corresponding to S is said to be *legal* if the spans of all *sdi*'s not occurring in S are within the architecturally imposed limits for short form instructions. □

On the PDP-11, the span s of any extended branch instruction which is given a short translation must satisfy the condition $-254 \leq s \leq 256$.

2. Some Natural but Suboptimal Algorithms

Let us begin by considering some natural ways to assemble programs containing *sdi*'s. We shall restrict ourselves to assemblers which employ a classical multi-pass organization. This is, they first read the program, producing a symbol table of labels and corresponding relative addresses. Then, during a subsequent pass over the program, they translate the program into actual machine code. The basic question here is which addresses should be assigned to those labels which have been preceded by one or more *sdi*'s. In the sequence

```
      jbr  L
A:    α
L:
```

the relative address of the label A cannot be determined until after the size of the code sequence α has been determined. If α translates into at most 254 bytes of code then the **jbr** can be translated into a **br** and A can be assigned relative address 2. Otherwise, the **jbr** becomes a **jmp** and A is assigned relative address 4.

It is tempting to circumvent this difficulty by using a finite sized buffer to look ahead some fixed amount before deciding how to translate a given *sdi.* The inadequacy of this approach is shown by the program appear-

[2] Throughout this paper the phrase "program length minimization" means minimizing the total length of the span-dependent instructions of the program in question without reordering the code.

ing in Figure 1. If the code segment α involves no more than two bytes of code then *all* the **jbr**'s can be translated as **br**'s, otherwise they must all be translated as **jmp**'s. The point here is that no bounded amount of lookahead will guarantee the optimal assignment of addresses to labels.

Another possible approach is to assign maximum addresses to labels during pass 1; that is, assume that each *sdi* must be translated to its long form. One or more intermediate passes can then be used to adjust downward the sizes of any *sdi*'s which can be given short translations (according to the current symbol table contents) and update the symbol table values of the program's labels. The actual translation to machine code is then done during a final pass. As an example of this technique let us again consider the program in Figure 1 and assume that the code sequence α produces exactly 2 bytes of machine code. Recall that under these conditions all **jbr**'s of the program can be translated to **br**'s. During the first pass, all the **jbr**'s will be treated as having a size of 4 bytes resulting in L_i being assigned address $4 + 256 * i$ for $0 \leq i < n$. Label L_n will be assigned address $2 + 256 * n$. During the second pass the span of the statement "**jbr** L_n" will be discovered to be 256. This allows the "**jbr** L_n" statement to be tested as a **br** instruction and decreases the assigned addresses of both L_{n-1} and L_n by 2. During each subsequent pass one more **jbr** is discovered to be translatable to a **br** and the symbol table values of the L_i's reach their final values after $n + 1$ passes.

In practice, very few programs exhibit the control structure shown in Figure 1. It would therefore probably be sufficient to restrict the process to but one intermediate pass. This is in fact the approach taken by the UNIX assembler. The resulting assembly algorithm can be expected to give near optimal performance. It does, however, require three complete passes[3] over the source code. Moreover, even if passes were continued until convergence occurred, the method would still fail to minimize the size of programs such as the one appearing in Figure 2, for which the algorithm would converge with X having relative address 0 and Y having relative address 262. Thus, both *sdi*'s would have to be translated to their long form. However, it is easy to see that if X and Y are assigned addresses 0 and 256 respectively, then the short form can be used for both instructions. It should be noted that the control structure depicted in Figure 2 corresponds to the standard implementation of a "while" loop and, as such, is not unexpected.

The point here is that multipass algorithms which start with the maximum address assignment and attempt to converge downward to a final solution must be suboptimal, although they can be expected to give good performance if terminated after a few iterations. Multipass algorithms which start with the minimum address assignment and converge upwards give optimal performances but cannot be terminated until convergence occurs.

Fig. 1. The inadequacy of the "window" approach.

```
                jbr          L1
L0:             Δ = 252
                jbr          L2
L1:             Δ = 252
                .
                .
                .
                jbr          Li
Li-1:           Δ = 252
                .
                .
                .
                jrb          Ln
Ln-1:           Δ = 252
                α
Ln:
```

Fig. 2. A bistable configuration.

```
X:              jne          Y
                Δ = 252
                jbr          X
Y:
```

Finally, note that the convergence of multipass algorithms cannot be guaranteed if the operands of *sdi*'s are allowed to be unrestricted expressions. To see this, consider the program fragment in Figure 3. The operands of the **jbr**'s are such that choosing a short translation for a given **jbr** forces some other **jbr** to be given a long translation. Conversely, whenever a **jbr** is treated as a **jmp**, it becomes possible to convert some other **jbr** to a **br**. The only stable situation is to translate all three **jbr**'s as **jmp**'s. Unfortunately, a multipass algorithm which attempts to converge toward a solution will fall victim to some variant of the following cycle of reasoning:

1) Since the **jbr** at A is long, the **jbr** at C can be short;
2) Since the **jbr** at C is short, the **jbr** at B must be long;
3) Since the **jbr** at B is long, the **jbr** at A can be short;
4) Since the **jbr** at A is short, the **jbr** at C must be long;
5) Since the **jbr** at C is long, the **jbr** at B can be short;
6) Since the **jbr** at B is short, the **jbr** at A must be long;
7) repeat from 1).

Thus, it is far from clear precisely what strategy should be used by a multipass algorithm in order to meet the twin requirements of convergence and optimal performance.

3. An Optimal, Two-Pass Algorithm

The algorithm to be presented in this section is essentially an iterative algorithm which starts with an empty selection set and converges upwards to a final selection set corresponding to the shortest legal transla-

[3] For example, the first pass of the UNIX assembler assigns "estimated" addresses to labels, the second pass determines which form should be used for each *sdi* and assigns "final" addresses to labels, and the third pass does the actual translation to object code.

Fig. 3. Nonconvergence of iterative techniques.

```
A:      jbr     .+260 − (C − B)
B:      jbr     .+260 − (D − C)
C:      jbr     .+260 − (B − A)
D:
```

tion of the program in question. As we have just seen, the convergence of such an algorithm cannot be guaranteed unless some restrictions are placed on the operands of the *sdi*'s in the program. The restrictions which we need are provided in the following two definitions.

Definition: A *simple expression* is a relocatable expression containing exactly one label. □

Without loss of generality, we can consider all simple expressions to be of the form "label ± constant." Thus the expression "A + B − C", although relocatable, is not simple.

Definition: An *sdi* I is said to be *pathological* if there exist selection sets S_1 and S_2 such that, 1) S_1 is included in S_2, 2) the span of I under S_1 is outside the range allowed for a short form instruction, and 3) the span of I under S_2 is within the range allowed for a short form instruction. □

As an example of a pathological instruction, consider the statement labeled A in Figure 3 and the selection sets $S_1 = \{\}$ and $S_2 = \{A, B, C\}$. Under the address assignment implied by S_1, the span of the instruction A is 258, which is too large for a short form instruction. On the other hand, the span of A under S_2 is only 256, which is within the limits allowed for a **br** instruction.

The algorithm to be presented in this section requires that all *sdi*'s in a program be nonpathological and have simple operands. We do not feel that any significant loss of utility results from this requirement. Fewer and fewer computers today employ a uniform instruction size, and thus even simple expressions of the form "L ± c" are somewhat confusing and of limited usefulness.[4] Indeed, all of the instruction operands produced by the UNIX C-compiler are both simple and nonpathological. As we shall soon see, pathological instructions are so obfuscating that they can be proscribed solely on the grounds of program (un)clarity.

At this point it is necessary to describe how to enforce these restrictions. The simplest method for enforcement is to automatically translate any *sdi* which is either nonsimple or pathological to a long form instruction. This method will revoke the guarantee that the translated program is as short as possible. However, as we shall see in Section 4, the optimal translation of programs containing pathological instructions is prohibitively expensive anyway. Thus we feel that we are really sacrificing very little by discarding them at the onset.

Identifying nonsimple instructions is easy. In order to identify those remaining simple instructions which are pathological, we note the following.

LEMMA: Let I be an *sdi* with simple operand E. Let L be the label occurring in E. Let S be some selection set.

1) If I precedes L in the program, then the span of I according to S is monotonically nondecreasing as elements are added to S.

2) If I occurs after L in the program, then the span of I according to S is monotonically nonincreasing as elements are added to S.

PROOF: Since E is simple, I is of the form "**jbr** L ± c". The only *sdi*'s whose length affects the span of I are those lying strictly between I and L. If I precedes L, adding any of these intervening *sdi*'s to S can only increase the address of L while leaving the address of I unchanged. Thus the span of I, which is L ± c−., cannot decrease. 2) is proven in a similar manner. □

Since the spans of instructions with simple operands change monotonically with additions to the selection set, it is only necessary to look at the span of each *sdi* relative to the empty selection set and the selection set consisting of all *sdi*'s in the program. Call these the *empty span* and the *full span* respectively.

Let us return for a moment to the case of the PDP-11. Suppose that I is the *sdi* "**jbr** L ± c". If I precedes L, then I is pathological iff the empty span of I is <-254 and the full span is ≥ -254. If I occurs after L, then I is pathological iff the empty span of I is >256 and the full span of I is ≤ 256. (As a consequence of this, note that no simple operand having c = 0 can ever by pathological.) For example, consider the code fragment shown below.

```
X:      Δ = 50
Y:      jeq     someplace
        Δ = 50
Z:      jbr     X + 360
```

The span of the instruction labeled Z is X + 360 − Z. Letting y denote the size of the instruction labeled Y, Z = X + 100 + y, and thus the span of Z is 260 − y. If the instruction labeled Y is translated as **beq,** then the (empty) span of Z is 258 and hence Z must be translated as a **jmp.** On the other hand, if Y is translated as a **bne, jmp** pair, then the (full) span of Z is only 254 and Z can be translated as a **br.** Thus Z is pathological.

The code fragment used in the previous example is reminiscent of doing an indexed jump into a table (namely, X) of branch addresses. The operand X + 360 could presumably arise when a smart compiler folds a constant subscript. However, the numbers involved here suggest that the subscript was out of range! Thus we can regard this example as justifying our use of the adjective "pathological" to describe instructions such as Z above. Indeed, it is very difficult, if not impossible, to construct program schemes containing pathological instructions whose computation does *not* depend on the translation selected for its *sdi*'s.

At this point we have enough terminology to describe

[4] The traditional interpretation of an assembly-time expression demands that the expression "L + *c*" refer to that machine location which is *c storage units* beyond the instruction labeled "L". This location is usually not the same as the one which is *c instructions* beyond L.

our algorithm. Throughout the rest of this section let n denote the number of *sdi*'s in the program being assembled. We suppose further that all *sdi*'s have simple operands and are nonpathological.

During the first pass we assign addresses to instructions and build a symbol table of labels and their addresses according to the minimum address assignment. We do this by treating each *sdi* as having its shorter length. We also number the *sdi*'s from 1 to n in order of occurrence and record in the symbol table entry for each label the number of *sdi*'s preceding it in the program. Simultaneous with pass 1 we build a set

$S = \{(i,a,l,c) | 1 \leq i \leq n$, a is the minimum address of the ith *sdi*, l and c, are the label and constant components of the operand of the ith *sdi* respectively$\}$.

Between passes 1 and 2 we will construct an integer table LONG[l:n] such that LONG[i] is nonzero if and only if the ith *sdi* must be given a long form translation. Initially LONG[i] is zero for all i.

At the heart of our algorithm is a graphical representation of the interdependencies of the *sdi*'s of the program. For each *sdi* we construct a node containing the empty span of that instruction. Nodes of this graph will be referred to by the number of the *sdi* to which they correspond. Directed arcs are now added to the graph so that $i \rightarrow j$ is an arc if and only if the span of the ith *sdi* depends on the size of the jth *sdi*, that is, the jth *sdi* lies between the ith *sdi* and the label occurring in its operand. It is easy to see that the graph we have just described can be constructed from the information present in the set S and the symbol table.

The significance of this graph is that sizes can be assigned to the *sdi*'s of the program so that the span of the ith *sdi* is equal to the number appearing in node i if and only if all the children of i can be given short translations.

After the structure is built we process it as follows. For any node i whose listed span exceeds the architectural limit for a short form instruction, set LONG[i] equal to the difference between the long and short forms of the ith *sdi*. Increment the span of each parent of i by LONG[i] if the parent precedes the child in the program. Otherwise, decrement the span of the parent by LONG[i]. Finally, remove node i from the graph. Clearly this process must terminate. Any nodes left in the final graph correspond to *sdi*'s which can be translated in the short form.

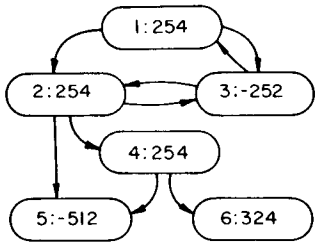
Now construct a table INCREMENT[0:n] by defining INCREMENT[0] = 0 and INCREMENT[i] = INCREMENT[$i - 1$] + LONG[i] for $1 \leq i \leq n$. INCREMENT[i] represents the total increase in size of the first i *sdi*'s in the program. At this point we can adjust the addresses of each label L in the symbol table. If L is preceded by i *sdi*'s in the program, then add INCREMENT[i] to the value of L in the symbol table. Finally, we do the traditional second assembly pass using the LONG table to specify how each *sdi* is to be treated.

As an example of the operation of this algorithm, consider the program shown in Figure 4. The initial dependency graph is shown in Figure 5. The numbers occurring inside each node are, respectively, its node number and current span. Initially nodes 5 and 6 are the only nodes whose spans necessitate translation as long form instructions. Node 5 is removed from the graph and the spans of both nodes 2 and 4 are increased to 256. Node 6 is now removed, changing the span of node 4 to 258. Since this is too large a span for a **br**, node 4 can be removed, increasing the span of node 2 to 260 (the long form of **jeg** is 6 bytes long). Finally, node 2 is removed, increasing the span of node 1 to 256 and decreasing the span of node 3 to −254, both of which are legal spans for short form instructions. The final graph is shown in Figure 6. Thus the length of the program appearing in Figure 4 is minimized by giving a short translation to the two statements "**jne** B" and "**jbr** A". All the other *sdi*'s must be given long translations.

Fig. 4. An example program for the algorithm.

```
          Δ = 100
A:        jne       B
          Δ = 248
          jbr       C
          jbr       A
B:        jeq       E
          Δ = 246
          jbr       A − 10
C:        jbr       F + 20
          Δ = 2
E:        Δ = 300
F:
```

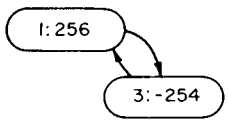
Fig. 5. The initial dependency graph.



The following facts provide the basis for the claim that the algorithm minimizes program size. (1) The decision to use a long translation for a given *sdi* is not made until a long translation for that instruction has been *proven* necessary. (2) Since *sdi*'s are required to be nonpathological, it will never be the case that once a given *sdi* has been shown to require a long translation, subsequent selections of long translations for other *sdi*'s will allow its translation to take the short form. (3) The order in which nodes are removed from the graph is irrelevant. Once the span listed in a node exceeds the allowed range for a short form instruction, the node must eventually be removed.

The running time of the algorithm just presented is

Fig. 6. The final dependency graph.



essentially proportional to the number of arcs in the dependency graph because no arc is ever followed from a child to a parent more than once. In the worst case there could be nearly n^2 such arcs but for reasonable programs it is likely that there would be only $O(n)$ arcs. Since space is likely to be of more concern than time in the implementation of this algorithm, we suggest that a brute-force scan of the set S be used to find the parents of a given node. The worst case running time of the algorithm[5] remains $O(n^2)$ but the space requirement is reduced to $O(n)$ because it is no longer necessary to explicitly construct the arcs of the dependency graph.

Mark Linton, an undergraduate student at Princeton, has implemented the algorithm in this final form and incorporated it into the UNIX assembler for the PDP-11 as a substitute for that assembler's intermediate pass. As a test of the effectiveness of our techniques, the entire UNIX operating system was compiled into PDP-11 assembly language and subjected to the two assemblers. The UNIX code contained 1424 *sdi*'s (all of which were simple and nonpathological) and assembled into approximately 27K bytes of instructions. The original assembler used a long form for 80 of these *sdi*'s whereas the modified (and optimal) assembler produced only 60 long translations. More significantly, the modified assembler ran 25% faster than the original assembler. Several points should be made here. First, since such a high percentage of *sdi*'s can be translated, short, extensive space savings (in this case about 4.5K bytes, or 17% of the total operating system size) can result from doing long/short optimizations. Second, the heuristic used in the original UNIX assembler is quite effective, coming in this case within 60 bytes of the minimal size. Third, the major justification for using the method proposed in this paper is the assembly time which can be saved while still producing optimal translations.

In closing this section, it should be noted that the requirement that programs not contain any pathological instructions was crucial for guaranteeing the optimal convergence of our algorithm. The requirement that all *sdi*'s have simple operands was made solely for convenience. We leave as an exercise for the reader the task of modifying the algorithm given above to accommodate programs containing nonsimple operands. It should also be noted that the processing described above is unnecessary in the case of those *sdi*'s which must be translated long under *any* selection set or which can be translated short under all selection sets.

[5] For each *sdi m* which requires a long translation we must adjust the span of all *sdi*'s whose span is dependent on *m*.

4. Minimizing Program Length is NP-Complete

In the previous section an efficient algorithm for the length minimization of programs with simple operands was presented. If a program is allowed to contain pathological or nonsimple operands, it is still possible to minimize its length, although the algorithm for doing so may no longer be efficient. Since there are only two ways of translating each *sdi* in a program, there are at most $O(2^n)$ possible translations of a program containing n *sdi*'s. We could simply generate each of these translations, discard those which are not legal, and select as our result the shortest surviving program. The algorithm we have just described would obviously require an enormous amount of time to process any program of substantial size. Nevertheless, it is our goal in this section to show that there really is no significantly better way to handle the general problem than the bruteforce technique we have just described. We shall do this by showing that the problem of determining the minimum feasible length of an arbitrary program is NP-complete.

The class of *NP-complete* problems consists of a large number of essentially combinatorial problems for which all known algorithms require (in the worst case) exponential time. Moreover, if any *one* of these problems could be solved in polynomial time, then they *all* could. Many famous optimization problems such as the traveling salesman problem and the chromatic number problem for graphs have been shown to be NP-complete. These problems have been studied for decades without the discovery of any efficient algorithmic solutions. It is therefore widely believed that no polynomial time solution exists for any NP-complete problem. Readers desiring more information about NP-complete problems may consult the literature (e.g. [1] or [4]) for additional background and terminology).

For technical reasons it is useful to require that all problems be phrased so as to have a yes/no answer. Accordingly, the problem considered in this paper can be cast as follows.

Definition: Given an integer s and an assembly language program P, the *instruction length assignment problem* is to determine whether P can be assembled to yield an object program of size s or less. □

We now need some sort of measure of the *size* of a problem. Intuitively, the size of a problem is that parameter (or parameters) of the problem which determines the running time of algorithms which solve the problem. In the case of the instruction length assignment problem, the size of an instance of the problem can be defined to be the number of *sdi*'s in a program.[6]

In order to show that a problem is NP-complete, it is necessary to show, first, that the problem can be solved nondeterministically in an amount of time which is bounded by a polynomial function of the problem size,

[6] Alternatively, the number of statements in a program could be defined to be the size of a problem. We view this definition of size as being less relevant to those aspects of the problem which interest us.

and secondly, that the problem is at least as hard as any other problem which is solvable in nondeterministic polynomial time. The first requirement is a technical detail which is usually easy to verify. Accordingly we shall not bother to demonstrate it in our proofs.

The second requirement is most easily fulfilled by showing how to *transform* an instance x of some known NP-complete problem X to an instance y of the problem of interest Y. This transformation must be efficient in the sense that it can be done in polynomial time (in the size of x) and does not increase the size of x by more than a polynomial factor. Moreover, the answer to x must be yes iff the answer to y is yes. The "known NP-complete problem" which we shall employ in our proofs is the following.

Definition: An instance of the 3-*satisfiability problem* is a set of *variables* $\{x_i | 1 \leq i \leq n\}$ and a collection of *clauses* $C_1, \ldots, C_m$ such that each C_j is a set consisting of exactly 3 *literals* of the form x_i or $\bar{x}_i$. A given instance of the problem is said to be *satisfiable* if there exists a truth assignment f mapping $\{x_i | 1 \leq i \leq n\}$ into {**true**, **false**} such that for every j in the range $1 \leq j \leq m$ there exists an i such that $f(x_i) =$ **true** and $x_i \in C_j$ or else $f(x_i) =$ **false** and $\bar{x}_i \in C_j$. The size of an instance is simply $n + 3m$. □

The 3-satisfiability problem was first defined and proven NP-complete in [2]. We can now state and prove the main result of this section.

THEOREM 1: The instruction length assignment problem is NP-complete if the assembly language programs under consideration are allowed to contain span-dependent instructions whose operands are arbitrary assembly-time expressions.

PROOF: We shall show how to transform an arbitrary instance I of the 3-satisfiability problem to a program P_I which can be translated to a certain minimum length iff I is satisfiable. For each variable x_i, P_I contains the code fragment

```
Yi:   jbr
      Δ = 254
Zi:
```

For every clause $C_j = \{l_{j,1}, l_{j,2}, l_{j,3}\}$, P_I contains the code fragment

```
Aj:   Δ = 246
      jbr     tj,1
      jbr     tj,2
      jbr     tj,3
Bj:
```

If $l_{j,k}$ is an unnegated literal x_i, then $t_{j,k}$ is ".+Z_i−Y_i". If $l_{j,k}$ is an negated literal $\bar{x}_i$, then $t_{j,k}$ is ".+Z_i−Y_i−512". Finally, for every clause C_j we add to P_I exactly $n + 3m + 1$ copies of the statement "**jbr** .+B_j−A_j". Thus P_I contains a total of $n + 3m + (n + 3m + 1)m$ *sdi*'s. The total length of a given translation of P_I is $256n + 254m + 2nm + 6m^2 + 2q$ where q is the number of **jmp** instructions in the translated version of P_I.

We shall now show that I is satisfiable iff P_I can be assembled to produce a program of size $258n + 260m + 2nm + 6m^2$ or less. Equivalently, I is satisfiable iff P_I can be assembled to an object program containing at most $n + 3m$ **jmp** instructions.

The following observation will be useful. Suppose that c is an even integer constant in the range $0 \leq c \leq 258$. The statement "**jbr** .+c" can be given a short translation iff $c \leq 256$. The statement "**jbr** .+c−512" can be given a short translation iff $c = 258$.

In any translation of P_I, each of the statements "Y_i:**jbr**." can be translated as either "Y_i:**br**." or as "Y_i:**jmp**.". Thus, for $1 \leq i \leq n$, the distance $Z_i - Y_i$ can be either 256 or 258. These n selections correspond to an assignment of truth values to the variables of I in a natural way, namely, $Z_i - Y_i$ is 256 iff x_i is true in the given truth assignment.

In any translation of P_I, each of the distances $B_j - A_j$ must be 252, 254, 256, or 258. The first 3 possibilities arise when at least one of the **jbr**'s between A_j and B_j is given a short form translation. This is, of course, equivalent to saying that at least one of the literals of C_j is made true by the truth assignment determined by the $Z_i - Y_i$ distances.

If I is satisfiable, it is possible to select the $Z_i - Y_i$ distances with $B_j - A_j \leq 256$ for $1 \leq j \leq m$. This means that *all* of the statements "**jbr** .+$B_j - A_j$" can be given short translations. Thus no more than $n + 3m$ **jbr**'s need be translated as **jmp**'s.

Conversely, if I is unsatisfiable, then for any choice of the $Z_i - Y_i$ distances, there will be at least one value of j for which all three of the **jbr**'s between A_j and B_j must be given long translations. This means that $B_j - A_j$ will be 258 and so all $n + 3m + 1$ of the statements "**jbr** .+B_j−A_j" must be translated as **jmp**'s. Thus P_I will contain at least $n + 3m + 1$ **jmp**'s. This completes the proof of the theorem because it is obvious that the transformation of I to P_I can be done in polynomial time. □

It turns out that the computational difficulty in minimizing program size is due to the presence of pathological instructions and is independent of whether operands are constrained to be simple. We show this below.

THEOREM 2: The instruction length assignment problem is NP-complete even if all operands of *sdi*'s are simple.

PROOF: Let I be an instance of the 3-satisfiability problem having n variables x_i, $1 \leq i \leq n$, and m clauses $C_j = \{l_{j,1}, l_{j,2}, l_{j,3}\}$. Let $q = n + m + 1$. We shall construct a program Q_I which can be legally translated with a selection set of size $n + 2m$ or less iff I is satisfiable.

First, consider the program P_I shown in Figure 7. As in the proof of Theorem 1, the translations selected for the statements labeled Y_i will correspond to a unique assignment of truth values to the variables x_i of I. Specifically, Y_i is translated as a **br** (making $Z_i - Y_i = 2$) if $x_i =$ **true**; Y_i is translated as a **jmp** (making $Z_i - Y_i = 4$) if $x_i =$ **false**.

Fig. 7. Basic construction for proof of Theorem 2.

Y_1:	**jbr**	·
Z_1:	**jbr**	·
	α_1	
	·	
	·	
	·	
Y_n:	**jbr**	·
Z_n:	**jbr**	·
	α_n	
$A_{1,1}$:	**jbr**	$A_{1,1} + t_{1,1}$
$A_{1,2}$:	**jbr**	$A_{1,1} + t_{1,2} + 4$
$A_{1,3}$:	**jbr**	$A_{1,1} + t_{1,3} + 8$
	β_1	
	·	
	·	
	·	
$A_{m,1}$:	**jbr**	$A_{m,1} + t_{m,1}$
$A_{m,2}$:	**jbr**	$A_{m,1} + t_{m,2} + 4$
$A_{m,3}$:	**jbr**	$A_{m,1} + t_{m,3} + 8$
	β_m	

The $t_{j,k}$ are expressions defined as follows. If $l_{j,k}$ is an unnegated literal, say x_i, then $t_{j,k}$ is "$Z_i - Y_i + 254$". If $L_{j,k}$ is a negated literal, say $\bar{x}_i$, then $t_{j,k}$ is "$Z_i - Y_i - 258$". The $t_{j,k}$ have been carefully selected so that

1) $A_{j,1}$ is translatable as a **br** iff the literal $l_{j,1}$ is made **true** by the truth assignment defined by the $Z_i - Y_i$ distances.
2) If $A_{j,1}$ is translated as a **jmp**, then $A_{j,2}$ is translatable as a **br** iff the literal $l_{j,2}$ is made **true** by the truth assignment defined by the $Z_i - Y_i$ distances.
3) If $A_{j,1}$ and $A_{j,2}$ are both translated as **jmp**'s then $A_{j,3}$ is translatable as a **br** iff the literal $l_{j,3}$ is made **true** by the truth assignment defined by the $Z_i - Y_i$ distances.

It is easy to verify that

(*) I is satisfiable iff P_i can be translated so that for every j, exactly one of $A_{j,k}$, $1 \le k \le 3$, is translated as a **br** and the other two are translated as **jmp**'s.

The code segments denoted by α_i and β_j are called *enforcers* and have the following properties:

a) Each enforcer consists of exactly $2q + 2$ **jbr** statements.
b) Within any enforcer, if any one **jbr** is translated as a **jmp**, then at least $2q$ of the **jbr**'s in that enforcer must be translated in the long form. When this happens, we say that the enforcer *explodes*.
c) If the translation of Y_i and Z_i produces exactly 6 bytes of code (i.e. one **jmp** and one **br**), then all the **jbr**'s in α_i can be translated as **br**'s.
d) If the translation of Y_i and Z_i does not produce exactly 6 bytes of code (i.e. both are translated as **br**'s or else both are **jmp**'s), then α_i explodes.
e) Similarly, β_j must explode if the three statements $A_{j,k}$, $1 \le k \le 3$, do not produce exactly one **br** and two **jmp**'s.

It should be clear from the properties of enforcers and (*) above that

(**) I is satisfiable iff P_I can be translated to have exactly $n + 2m$ **jmp**'s.

To see this, first suppose that I is satisfiable and consider any one of the translations whose existence is implied by (*). The translations of the statements labeled Z_i can be freely chosen so that Z_i is a **jmp** iff Y_i is a **br**. By property (c) above, all of the **jbr**'s in the α_i's can therefore be translated as **br**'s. Moreover, by property e) above, all of the **jbr**'s in the β_j's can be translated as **br**'s. This gives us a total of exactly $n + 2m$ **jmp**'s.

Now suppose that a translation of P_I having exactly $n + 2m$ **jmp**'s exists. Clearly, none of the enforcers in such a translation could be exploded, for if it were, then the translation would contain at least $2q = 2n + 2m + 2 > n + 2m$ **jmp**'s. Thus, for every j, exactly one of the statements $A_{j,k}$, $1 \le j \le 3$, is a **br**. By (*), I is satisfiable.

Unfortunately, the operands in P_I are not all simple expressions. We must therefore show how to modify P_I to form a new program Q_I which has only simple operands and which can be translated to have $n + 2m$ **jmp**'s iff P_I can.

One of the main functions of the enforcers in P_I is to bind certain labels to fixed relative addresses in any minimal length translation of P_I. More precisely,

(***) A given translation of P_I contains exactly $n + 2m$ **jmp**'s iff label Y_i is at relative address $(4q + 10)(i - 1)$, $1 \le i \le n$, and $A_{j,1}$ is at relative address $(4q + 10)n + (4q + 14)(j - 1)$, $1 \le j \le m$.

The only operands appearing in P_I which are not simple occur in the statements labeled $A_{j,k}$. These operands are all of the form $A_{j,1} + Z_i - Y_i \pm c$ where c is some integer constant. This can, of course, be written as $Z_i + (A_{j,1} - Y_i \pm c)$. Notice that the subexpression in parentheses is absolute. By (***), this subexpression can be replaced with the constant $(4q + 10)(n + 1 - i) + (4q + 14)(j - 1) \pm c$. The resulting program is the desired Q_I. It is easy to verify that properties a) through e) of enforcers, as well as statements (*), (**), and (***), are also true of Q_I. Thus Q_I can be translated to contain $n + 2m$ **jmp**'s iff I is satisfiable.

In order to complete our proof, we must show how to construct an enforcer having only simple expressions as operands. Figure 8 shows an enforcer which explodes to produce at least $2q$ **jmp**'s in any legal translation in which γ produces other than k bytes of code. Observe first that

1) If A_1 is translated as a **jmp**, then B_1 must be too.
2) For $1 \le i < q$, if B_i is translated as a **jmp**, then B_{i+1} must be too.
3) If B_q is translated as a **jmp**, then A_q must be too.
4) For $1 \le i < q$, if A_{i+1} is translated as a **jmp**, then A_i must be too.

Thus all of the A_i's and B_i's must be translated as **jmp**'s if any one of them is. Moreover, if either F or G is

Fig. 8. Constructing an enforcer.

E:	γ	
F:	**jbr**	E + 256 + k
G:	**jbr**	E − 252 + k
A_1:	**jbr**	B_2 + 250
B_1:	**jbr**	F − 248
A_2:	**jbr**	B_3 + 250
B_2:	**jbr**	B_1 − 250
	.	
	.	
	.	
A_i:	**jbr**	B_{i+1} + 250
B_i:	**jbr**	B_{i-1} − 250
	.	
	.	
	.	
A_{q-1}:	**jbr**	B_q + 250
B_{q-1}:	**jbr**	B_{q-2} − 250
A_q:	**jbr**	C + 252
B_q:	**jbr**	B_{q-1} − 250
C:		

translated as a **jmp**, then B_1 (and hence all the A_i's and B_i's) must be too.

Consider the statement "F: **jbr** E + 256 + k", which we could just as well write as "F: **jbr** .−|γ| + 256 + k". If |γ| < k, then k−|γ| + 256 > 256 and F must be translated as a **jmp** thus triggering the required explosion.

Next consider the statement "G: **jbr** E − 252 + k", which could just as well be written "G: **jbr** .−|γ| − f − 252 + k" where f is the length of the code produced by the statement F. Since $f \geq 2$, if |γ| > k, then k−|γ| − f − 252 < $-f$ − 252 ≤ −254. This means that G must be translated as a **jmp** and the enforcer must explode.

Hence any legal translation of the program fragment in Figure 8 must contain at least $2q$ **jmp**'s if |γ| ≠ k. We leave to the reader the task of verifying that if there exists a legal translation of this program fragment in which |γ| = k, then there exists a legal translation in which every one of the **jbr**'s shown is translated as a **br.** □

5. Summary

The problem of minimizing the length of programs containing span-dependent instructions was considered. An efficient algorithm was presented for minimizing the length of programs all of whose span-dependent instructions were nonpathological and had simple operands. Although it is possible to remove the restriction to simple operands, the restriction to nonpathological instructions is apparently essential. This was demonstrated by the proof that the instruction length assignment problem for programs with simple operands is NP-complete if pathological instructions are allowed.

Acknowledgments. Susan Graham, Mark Linton, and the referees provided several helpful comments on an early draft of this paper. An algorithm similar to the one presented in Section 2, but restricted to the case where the operands of *sdi*'s are labels only, has been independently discovered and implemented in the BLISS-11 compiler [8].

Received January, 1977; revised July, 1977

References

1. Aho, A.V, Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading Mass., 1974.
2. Cook, S.A. The complexity of theorem proving procedures. *Proc. 3rd Annual ACM Symp. on Theory of Comptng,* May 1974, 151–158.
3. Frieder, G., and Saal, H.J. A process for the determination of addresses in variable length addressing. *Comm. ACM 19,* 6 (June 1976), 335–338.
4. Karp, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972.
5. PDP-11 Processor Handbood. Digital Equipment Corp., Maynard, Mass., 1975.
6. Richards, D. L. How to keep the addresses short. *Comm. ACM 14,* 5 (May 1971), 346–349.
7. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17,* 7 (July 1974), 365–375.
8. Wulf, W., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O., and Geschke, C.M. *The Design of an Optimizing Compiler.* American-Elsevier, New York, 1975.