# Global Optimization of Microprograms Through Modular Control Constructs

Graham Wood

Dept of Computer Science
University of Edinburgh
Scotland

## Abstract

By taking due regard of the disciplines of structured programming, we extend the scope of optimization of microprograms from blocks of micro-operations delimited by labels and branches (straight line segments) to complete microprograms.

## Introduction

Heretofore, with the exception of Dasgupta's heuristic attempts to recognize symmetric pairs [1], the search for potential concurrency between micro-operations for the purpose of microprogram optimization has been confined to these micro-operations which occupy the same straight line segment of a sequentially expressed microprogram [2-6], ie. between micro-operations not separated by a branch or a merge in the notional flow of control of the microprogram. The reasons for this restriction are obvious enough: two micro-operations may be executed concurrently in an implementation only if the specified behaviour for the microprogram indicates that the execution of one is a necessary and sufficient condition for the execution of the other, a condition which may not easily be verified when the two are separated by a label or a branch operation - a critical point - in the microprogram.

But membership of the same straight line segment is not a necessary condition for two micro-operations to be potentially concurrent. Consider the flow of control represented in figure 1. If micro-operation D is data independent of all of the operations on both branches of the path between B and C, and D is data independent of A, then A and D may be executed concurrently.

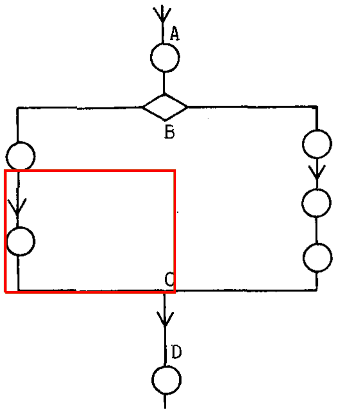

Figure 1

This is precisely the pattern of control flow which is associated with modular control constructs, the fundamental primitives of structured programming. If we can carry through into the implementation phase the modular structure in which microprograms should be designed, then we shall be able to exploit that structure in enhanced potential for concurrency between the micro-operations. The rest of this paper describes a system which endeavours to do just that.

## Background

We introduce MDL, Microprogram Design Language, whose statements are either register transfer expressions or control directives. The control constructs supported in the language are:-

(1) if COND goto LABEL
(2) if COND then ........(else).........finish
(3) (while COND) loop.......repeat (while COND)
(4) exit-n   (jump out of n levels
        of nested loop)
(5) wait until COND
(6) Subroutine call and return.

A microprogram is expressed in MDL as a sequential list of statements (hereafter referred to as micro-operations). Detection of potential concurrency between the micro-operations for the purpose of generating an optimized implementation

CH1516-4/79/0000-0001$00.75 © 1979 IEEE

normally is performed automatically, although the language does permit the designer to specify synchronization explicitly. Since MDL is an implementation independent language, we are concerned here solely with the detection of logical potential concurrency between micro-operations. That is, we do not take into account at this stage the physical resources required to realize the micro-operations, although the method would not be altered significantly in doing so.

The issues involved in the determination of potential concurrency between micro-operations in the same straight line segment are well-known and amply described in the literature [2-6]. We content ourself here with summarizing the three situations where micro-operation B should be marked as being data dependent on micro-operation A, where A precedes B in the straight line segment. These are:

DD1: A and B both write to the same register.
DD2: B reads from a register which A writes to.
DD3: B writes to a register which A reads from.

We note in passing that condition DD3 in some implementations is a weak dependency relationship in which A and B may be activated from the same microinstruction word, but we shall not pursue that issue here.

## The Problem

What we are concerned about in generating a maximally parallel representation of a microprogram expressed in MDL is how to preserve the behaviour of the microprogram as defined, while not constraining to straight line segments our search for potential concurrency. That is, how do we preserve the indivisibility and inviolability of conditional block structures while optimizing over their boundaries? We wish to allow micro-operations to 'migrate' over a conditional block, but we do not wish to let them 'land' inside the block, nor do we wish to allow any micro-operation to migrate beyond the confines of its own block.

In describing how this is accomplished, we shall ignore for the present the problems created by critical points, concentrating initially only on the well structured control operations, to return later to explicit labels and branches.

## The Solution

We take it for granted that it is a straightforward task to analyse an MDL description to detect data dependency between micro-operations according to rules DD1 to DD3 above and we do not dwell on this aspect, save to point out that our implementation performs this task in time linearly proportional to the number of statements in the microprogram.

The problem lies in determining the control dependency which will ensure a correct implementation.

The observation that a micro-operation must be data independent of all component micro-operations of a conditional block before it is allowed to migrate over that block leads us to develop the concept of a hierarchy of levels of micro-operations within the microprogram. We now say that a micro-operation may be of two basic types: a primitive micro-operation which is an atomic entity, or a block-type micro-operation which may be expanded at one level lower to a set of component micro-operations. These component micro-operations may, in turn, be either primitive or block type.

We associate with each 'if...then...else' block and loop ('wait until' COND is simply a degenerate loop) in the microprogram a block-type micro-operation with the constituent micro-operations of the control structure as the components of the block and we then insist that data dependency must be defined between micro-operations at the same level. For example, if a primitive micro-operation, B, outside a loop is found to be data dependent on a primitive micro-operation, A, inside the loop, then we mark B as being dependent on the block-type micro-operation representing the loop itself, thus preventing B from migrating into the loop. But this is not enough, as we see if we consider B above as itself being inside a loop. Then A and B are at the same level, but we still must prevent the whole loop containing B to be implemented until after the whole loop containing A has been.

In order to ensure complete inviolability and indivisibility of control blocks we demand adherence to the following rule:

Multi-level Dependency Rule: If A and B are primitive micro- operations with A data dependent on B, then the outermost block containing A but not B should be marked as dependent on the outermost block containing B but not A.

This rule guarantees that no block-type micro-operation will be packed into microinstruction words until all of its components' parents have already been packed and that no block-type micro- operation will be packed before the previous one at the same level has been completely packed (unless the two are independent).

The loop construct is a single entity, but the 'if...then...else' control structure is more complex. It consists of a single conditional micro-operation followed by two mutually independent sub-blocks. We model this by representing the whole conditional block as a single block-type micro-operation with three components: a primitive micro-operation, which is the condition at the head of the block, and two block-type micro-operations representing the 'if' block and 'else' block respectively. The Ifblock and Elseblock are each marked as dependent on the primitive conditional micro-operation. This serves to preserve the indivisibility of the total construct while keeping separate the components of the block and maintaining their relative interdependence.

Figure 2a represents a skeletal MDL microprogram and figure 2b represents a set of data dependency relationships which might exist between the micro-operations of 2a. Figure 2c illustrates the multi-level dependency graph generated to represent the microprogram with those relationships. (Each micro-operation is represented by a box with the level displayed in the top right corner of the box.)

```
(1)
loop
   (2)
   (3)
   if Cond then(4)
      (5)
      (6)
   else
      (7)
   finish
   while COND loop(8)
      (9)
      (10)
   repeat
   (11)
repeat while COND(12)
(13)
(14)
```

figure 2a

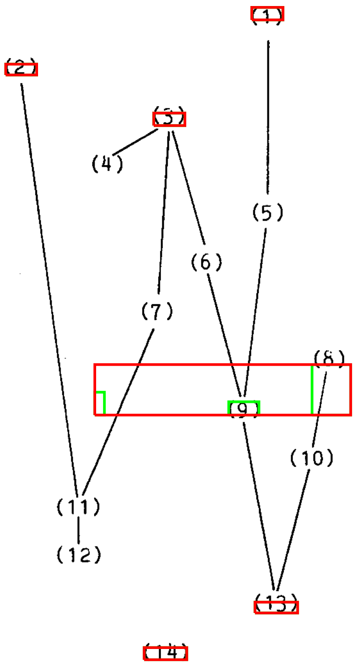

figure 2b

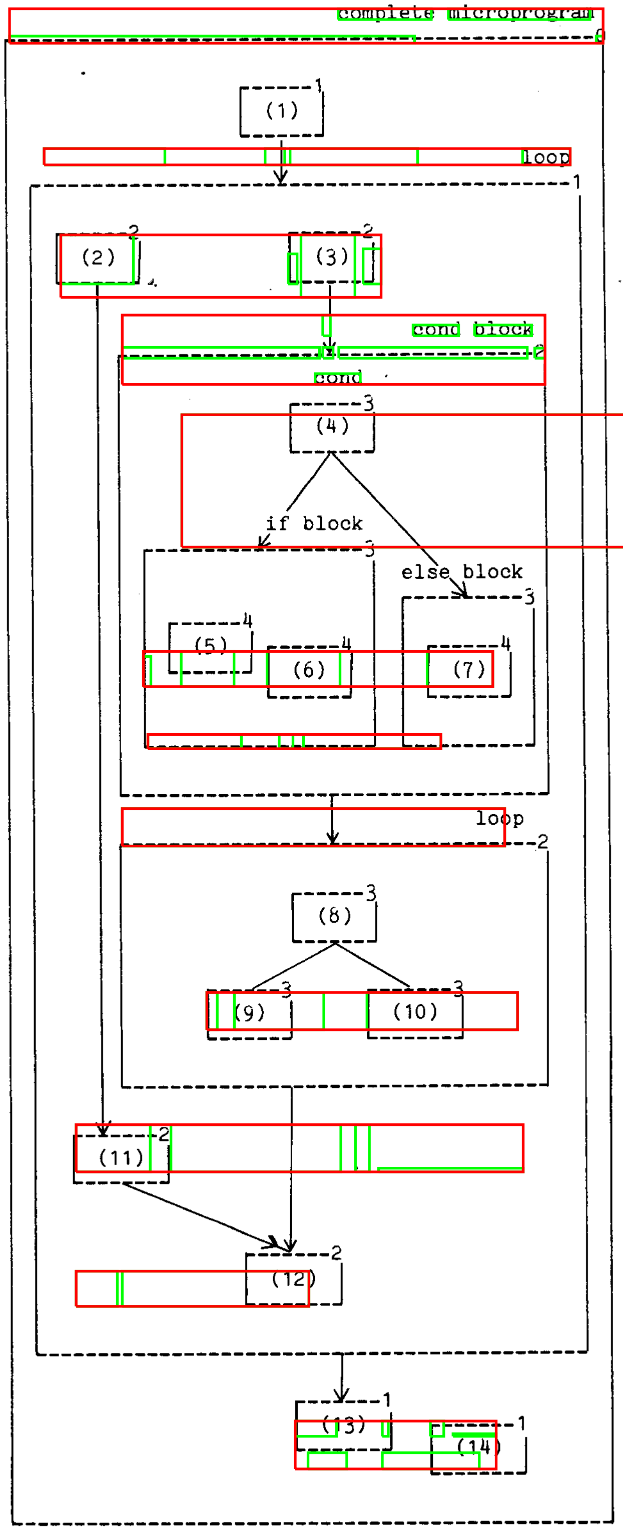

figure 2c

3

## Critical Points

We now turn our attention to the consideration of critical points: explicit branch operations and labels. Critical points disrupt the well-structured flow of control in an MDL description. Their merit or otherwise have been debated hotly in the context of structured programming, but we include such constructs in the language, without advocating their usage, in the belief that for the sake of generating as efficient microcode as desired any microprogramming language should be capable of expressing any operation which may be performed at the microprogram level (thereby depriving microprogrammers of any excuse for not using the language).

Unless we are prepared to trace the actual pattern of control flow associated with explicit labels and branches, critical points present absolute barriers to the migration of micro-operations. Dasgupta [1] has demonstrated that it is not practicable in any general way to attempt to optimize by exploiting the tracing of control flow like this, even where no modular control constructs are available and all control has to be effected via branches and labels. The provision of modular control constructs in MDL implies that any use of explicit labels and branches in a microprogram will almost certainly be to effect unstructured control flow, where indeed the critical points do present absolute barriers to code migration. We therefore take the view that the complexity of tracing and exploiting the pattern of control flow associated with explicit labels and branches in MDL far outweighs the negligible gains to be had from so doing. Thus critical points effectively partition an MDL microprogram into disjoint segments: not straight line segments, since the critical points may occur inside modular control blocks. It would be feasible simply to carve up the microprogram into these segments and to treat each one separately, but we seek a more elegant solution consistent with our handling of the other dependency relationships.

To that end, we associate with each critical point a block-type micro-operation whose components are all these micro-operations following the critical point, up to the first subsequent block terminating statement (ie. 'else', 'finish', or 'repeat') which serves to terminate the critical point block as well as the enclosing conditional block. (It is necessary to limit the range of critical point blocks in this fashion in order to maintain consistency of level between these micro-operations before and after the enclosing conditional block - a necessity when dealing with 'Exit' statements, as will be discussed below.)

The critical point micro-operation is marked as dependent on all preceding, otherwise childless, micro-operations at the same level as itself, thereby preventing the downward migration of these past the critical point. Those micro-operations which are components of the critical point block automatically are prevented from upward migration, but we must still build in safeguards against the upwards migration of succeeding micro-operations which are not themselves components of the critical point block, being outside the enclosing conditional block. We do this whenever dependency is being determined according to the Multi-level Dependency Rule described above by checking whether a critical point intervenes between the child and the parent of the relationship. If so, we substitute the critical point block for the parent and proceed with evaluating dependency according to the rule, thus maintaining consistency of approach for all types of dependency.

'<u>Exit</u>' micro-operations (jumps out of loops) act as critical points in a limited vicinity: that vicinity being between the 'exit' statement itself and the tail of the loop ultimately being exited from. These points delimit the scope of all micro-operations contained within that range and present an inviolable block to micro- operations without, but do not prevent the migration of micro-operations over the loop enclosing them, as a critical point would do. Figures 3a, 3b and 3c below illustrate the effect of an 'exit' statement in three situations. Figure 4 presents the dependency graph generated after the alteration of figure 2a through the addition of an explicit label to statement (6) and the changing of statement (10) to an exit out of two levels of loop.

```
loop
  .
  .
  .
  .
  exit
  .
  .
 (X)
  .  Range of X
  .
  .
  .
repeat
```

figure 3a

```
loop
  .
  .
  loop
    .
    .
    .
    exit-2
    .
    .
  repeat
  .
  .
 (X)
  .
  .  Range of X
  .
repeat
```

figure 3b

```
loop
  .
  .
  loop
    .
    .
    exit-2
    .
    .
  repeat
  .
  .
  .
repeat
.
(X)-Not affected
     by exit
```

figure 3c

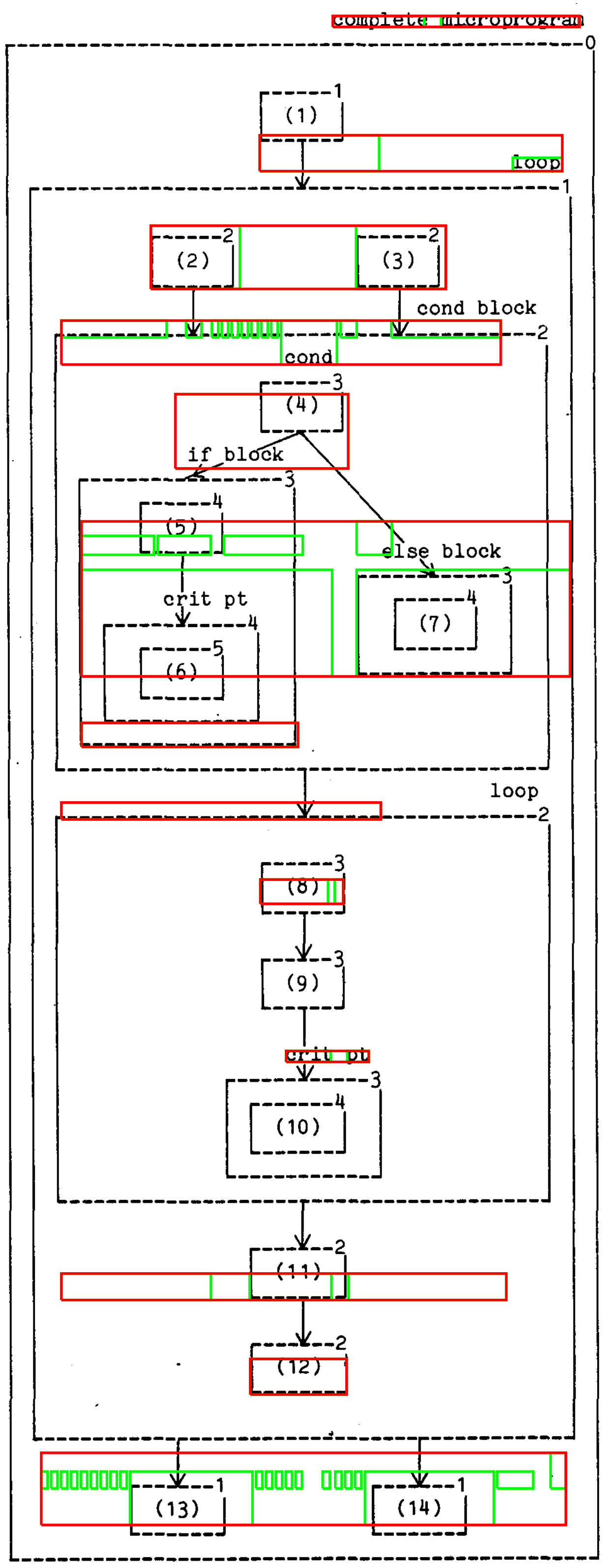

figure 4

There are many other subtle points of detail to be taken into account in generating a maximally parallel representation of an MDL description, but all operate within the context of the broad rules described above and none violate the elegance of the solution.

## Summary

We have described what we believe to be an elegant solution to the problem of global optimization of microprograms by taking advantage of the well-defined structure associated with modular control constructs. We have conceived the notion of a hierarchy of levels associated with the control blocks in a microprogram and have devised a rule for guaranteeing consistency between the levels. It is to be hoped that this increased scope for optimization will encourage the use of structured languages for microprogram design.

## References

[1] S. Dasgupta
"Parallelism in Loop-free Microprograms"
in Proc. IFIP Congress
Information Processing 77
B. Gilchrist (Ed)
North-Holland, Amsterdam 1977, pp 745-750

[2] S. S. Yau; A. C. Schowe; M. Tsuchiya
"On Storage Optimization of Horizontal Microprograms"
Proc. 7th Annual Workshop on Microprogramming,
Palo Alto, Ca. (Sept-Oct 1974) pp 98-106

[3] M. Tsuchiya; M. J. Gonzales
"An Approach to the Optimization of Horizontal Microprograms"
Proc. 7th Annual Workshop on Microprogramming,
Palo Alto, Ca. (Sept-Oct 1974) pp 85-90

[4] S. Dasgupta; J. Tartar
"Automatic Identification of Maximal Parallelism in Straight Line Microprograms"
IEEE T-C C-25, No 10 (Oct 1976) pp 986-991
See also comments on abovein IEEE T-C C-27,
No 3 (March 1978)

[5] P. Mallett
"Methods of Compacting Microprograms"
Ph.D. Thesis
Dept of Computer Science
University of Southwestern Louisiana
Lafayette, La. (Dec 1978)

[6] G.Wood
"On the Packing of Micro-operations into Microinstruction Words"
Proc. 11th Annual Workshop on Microprogramming,
Pacific Grove, Ca. (Nov 1978) pp 51-55